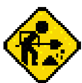
Curso de Engenharia Elétrica
Informática Aplicada à Engenharia Elétrica I



Apostila de ANSI C

Prof. Fernando Passold



Observação: material ainda está em fase de edição!
Usando sistema MiKTeX/LaTeX 2_ε de edição
Referência: http://www.miktex.org/
Prof. Fernando Passold <*fpassold@upf.br*>
Engenharia Elétrica
Semestres: 2005.1, 2006.1
Última atualização: 13 de março de 2006.



Sumário

Tópicos Previstos

1. Introdução à Lógica de Programação
   (a) Lógica
   (b) Seqüência Lógica
   (c) Instruções
   (d) Algoritmo
   (e) Programas
   (f) EXERCÍCIOS
2. Desenvolvendo algoritmos
   (a) Pseudocódigo
   (b) Regras para construção do Algoritmo
   (c) Fases
   (d) Exemplo de Algoritmo
   (e) Teste de Mesa
   (f) EXERCÍCIOS
3. Diagrama de Bloco (Fluxogramas)
   (a) O que é um diagrama de bloco?
   (b) Simbologia
   (c) EXERCÍCIOS
4. Constantes, Variáveis e Tipos de Dados
   (a) Constantes
   (b) Variáveis
   (c) Tipos de Variáveis
   (d) Declaração de Variáveis
   (e) EXERCÍCIOS
5. Comandos Básicos
   (a) Atribuição
   (b) Entrada
   (c) Saída
6. Operadores
   (a) Operadores Aritméticos
   (b) Operadores Relacionais
   (c) Operadores Lógicos
   (d) EXERCÍCIOS
7. Operações Lógicas
   (a) EXERCÍCIOS
8. Estruturas de Decisão e Repetição
   (a) Estruturas de Decisão
      i. SE ENTÃO / IF . THEN
      ii. SE ENTÃO SENÃO / IF . THEN . ELSE
      iii. Decisão Múltipla: CASO SELECIONE / SELECT . CASE
      iv. EXERCÍCIOS
   (b) Comandos de Repetição
      i. Repetição com Teste no Início: Enquanto x, Processar (Do While . Loop)
      ii. Repetição com Teste no Final: Até que x, Processar . (Do Until . Loop)
      iii. Processar ., Enquanto x (Do . Loop While)
      iv. Processar ., Até que x (Do . Loop Until)
      v. Repetição Automática (for)
      vi. EXERCÍCIOS
9. Arrays
   (a) Vetor
   (b) Matriz
   (c) Array Multidimensional
10. Manipulação De Strings
   (a) O Tipo De Dado String
   (b) Comandos de Entrada e Saída com Strings

Bibliografia Recomendada

Algoritmos:

1. ALGORITIMOS: teoria e prática. Rio de Janeiro: Campus, 2002. 916 p. ISBN 8535209263.
2. MANZANO, José Augusto Navarro Garcia; OLIVEIRA, Jayr Figueiredo de. Algoritmos: lógica para desenvolvimento de programação. 9.ed. São Paulo: Érica, 2000. 265 p. ISBN 857194329X.
3. MANZANO, José Augusto Navarro Garcia; OLIVEIRA, Jayr Figueiredo de. Estudo dirigido: algoritmos. 9.ed. São Paulo: Érica, 2004. 220 p. ISBN 857194413X.
4. SILVA, José Carlos G. da; ASSIS, Fidelis Sigmaringa G. de. Linguagens de programação: conceitos e avaliação : FORTRAN, C, Pascal, Modula-2, Ada, CHILL. São Paulo: McGraw-Hill, 1988. 213 p.
5. VILLAS, Marcos Vianna; VILLASBOAS, Luiz Felipe P.. Programação: conceitos, técnicas e linguagens. Rio de Janeiro: Campus, 1988. 195 p. ISBN 8570014775.
6. ZIVIANI, Nivio. Projeto de algoritmos: com implementações em Pascal e C. 3.ed. São Paulo: Pioneira, 1996. 267 p.

Linguagem C:

1. KERNIGHAN, Brian W.; RITCHIE, Dennis M.. C: a linguagem de programação. Rio de Janeiro: Campus, 1989. 208 p. ISBN 8570014104.
2. SCHILDT, Herbert; C Completo e Total, 3 ed., São Paulo: Pearson/Makron Books, 2004. 827 p.
3. SCHILDT, Herbert; GUNTLE, Greg. Borland C++ builder: the complete reference. Berkeley: Osborne McGraw Hill, 2001. 977 p. ISBN 0072127783.
4. SCHILDT, Herbert. C avançado: guia do usuário. 2.ed. São Paulo: McGraw-Hill, 1989. 335 p.
5. SCHILDT, Herbert. Programando em C e C++ com Windows 95. São Paulo: Makron Books, 1996. 537 p. ISBN 8534604797.
6. SCHILDT, Herbert. Turbo C avançado: guia do usuário. São Paulo: McGraw-Hill, 1990. 457 p.
7. SCHILDT, Herbert. Turbo C: guia de referência básica. São Paulo: McGraw-Hill, 1989. 242 p.
8. SCHILDT, Herbert. Turbo C: guia do usuário. 2.ed. São Paulo: McGraw Hill, 1989.
9. PUGH, Kenneth. Programando em linguagem C. São Paulo: McGraw-Hill, 1990. 251 p.
10. MANZANO, José Augusto Navarro Garcia. Estudo dirigido: linguagem C. 5.ed. São Paulo: Érica, 2001. 180 p. ISBN 8571944199.
11. DORNELLES FILHO, ADALBERTO AYJARA; SENAI. Fundamentos da linguagem C. 2.ed. Caxias do Sul: SENAI, 1998. 111 p.
12. HERGERT, Douglas. O ABC do Turbo C. São Paulo: McGraw-Hill, 1991. 341 p.
13. MIZRAHI, Victorine Viviane. Treinamento em linguagem C++. São Paulo: Makron Books, 1995. 2v. ISBN 8534602905 (v.1)
14. WIENER, Richard S. Turbo C: passo a passo. Rio de Janeiro: Campus, 1991. 319 p. ISBN 8570016123.
15. TREVISAN, Ricardo; LEHMANN, Anselm H.. Turbo C: guia de referência rápida para todas as funções. São Paulo: Érica, 1990. 141 p. ISBN 8571940525.

Parte I

Introdução à Programação

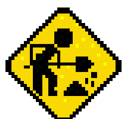
Curso de Engenharia Elétrica
Informática Aplicada à Engenharia Elétrica I



$1^{\underline{a}}$ Parte da Apostila de ANSI C

Introdução à Programação

Prof. Fernando Passold



Observação: Esta apostila está em constante fase de edição
Usando sistema de edição MiKTeX/LaTeX 2_ε para a mesma:
Ver: http://www.miktex.org/
Prof. Fernando Passold – Semestre 2005
Última atualização: 13 de março de 2006.



Capítulo 1

Introdução à Lógica de Programação

Contents

1.1 Programas e processos

Um computador nada mais faz do que executar programas. **Um programa é** simplesmente uma seqüência de instruções definida por um programador. Assim, um programa pode ser comparado a uma receita indicando os passos elementares que devem ser seguidos para desempenhar uma tarefa. Cada instrução é executada no computador por seu principal componente, o processador ou CPU (de unidade central de processamento).



Computador pensa? *Claro que não. O que faz é armazenar dados, dar forma a eles e informar o resultado. Como a fôrma dá forma aos ingredientes que formam o pudim. É por isso que informação não passa de dados que ganharam uma formatação. Mas será que a máquina pensou aí? Ainda não. Só processou.*
Quem costuma trabalhar em um terminal burro – o posto avançado de um computador central – conhece só meia verdade. A verdade inteira é que o computador na outra ponta também é burro. E se eu pensar que o computador pensa, pertenço à mesma família. A menos que aprenda que pensar é enxergar além de dados informes. Ou além da informação.
Quem se concentra demais nos dados e na informação só consegue enxergar o furo do pudim, onde não há sabor nem sensação. Todavia, quem pensa saliva. Estimulado pelas glândulas da intuição, desperta a imaginação que leva à criação. Já viu um micro com intuição?
Criar, programar, isto se faz com criatividade, a capacidade humana de enxergar nos dados informes aquilo que o computador não vê. Agostino D'Antonio, de Florença, trabalhou um grande bloco de mármore e não chegou a lugar nenhum. Desistiu da rocha ruim. Outros também. Por quarenta anos o bloco permaneceu em sua monolítica inanição até alguém enxergar nele uma possibilidade de criação. Na rocha informe

Michelângelo viu um Davi. E o esculpiu.
Por: Mario Persona, "Criando de cabo a rabo", janeiro - 2005, site: http://www.rhoempreendedor.com.br/.

Cada processador entende uma linguagem própria, que reflete as instruções básicas que ele pode executar. O conjunto dessas instruções constitui a **linguagem de máquina** do processador. Cada instrução da linguagem de máquina é representada por uma seqüência de bits distinta (dígitos 0's e 1's), que deve ser decodificada pela unidade de controle da CPU para determinar que ações devem ser desempenhadas para a execução da instrução.

Claramente, seria extremamente desconfortável se programadores tivessem que desenvolver cada um de seus programas diretamente em linguagem de máquina. Programadores não trabalham diretamente com a representação binária das instruções de um processador. Existe uma descrição simbólica para as instruções do processador – a **linguagem assembly** do processador – que pode ser facilmente mapeada para sua linguagem de máquina.

No entanto, mesmo assembly é raramente utilizada pela maior parte dos programadores que trabalha com linguagens de programação de alto nível – devido ao seu estilo "mnemônico". A figura 1.1 mostra à título de curiosidade as instruções assembly para programação de um **microcontrolador PIC 16Fxx** (note que são pouco mais de 35 instruções).

A figura 1.2 mostra um exemplo de programa (TESTER.ASM) em Assembly para PIF16Cxx que acende e apaga um LED. O LED está ligado ao bit 7 da porta B do microcontrolador. A tecla "1" é usada para acender o LED. A tecla "2" apaga o LED.

Em **linguagens de alto nível**, as instruções são expressas usando palavras, ou termos, que se aproximam daquelas usadas na linguagem humana, de forma a facilitar para os programadores a expressão e compreensão das tarefas que o programa deve executar. Várias linguagens de alto nível estão disponíveis para diversas máquinas distintas. Essa independência de um processador específico é uma outra vantagem no desenvolvimento de programas em linguagens de alto nível em relação ao uso de assembly. Em geral, cada linguagem de alto nível teve uma motivação para ser desenvolvida. **BASIC** foi desenvolvida para ensinar princípios de programação; **Pascal**, para o ensino de programação estruturada; **FORTRAN**, para aplicações em computação científica; **Lisp** e **Prolog**, para aplicações em Inteligência Artificial; **Java**, para o desenvolvimento de software embarcado e distribuído; e a **linguagem C**, para a programação de sistemas.

O fato de uma linguagem ter sido desenvolvida com uma aplicação em mente não significa que ela não seja adequada para outras aplicações. A linguagem C, juntamente com sua "sucessora" **C++**, é utilizada para um universo muito amplo de aplicações. Um dos atrativos da linguagem C é sua flexibilidade: um programador C tem à sua disposição comandos que permitem desenvolver programas com características de alto nível e ao mesmo tempo trabalhar em um nível muito próximo da arquitetura da máquina, de forma a explorar os recursos disponíveis de forma mais eficiente. Por este motivo, o número de aplicações desenvolvidas em C e C++ é grande e continua a crescer.

1.2 Desenvolvimento de software

Não há dúvidas hoje em dia quanto à importância do software no desenvolvimento dos mais diversos sistemas. Com esta evolução do papel do software em sistemas computacionais, veio também uma maior complexidade de programas e uma maior preocupação em desenvolver programas que pudessem ser facilmente entendidos e modificados (se necessário), não apenas pelo autor do programa mas também por outros programadores. A disciplina que estuda o desenvolvimento de programas com qualidade é conhecida como Engenharia de Software.

A **Engenharia de Software** estabelece alguns princípios de desenvolvimento que independem da linguagem de programação adotada. Estes princípios são utilizados nas três grandes fases da vida de um programa, que são a especificação, o desenvolvimento e a manutenção de programas. A especificação inicia-se com o levantamento de requisitos (ou seja, o que deve ser feito pelo programa) e inclui a análise do sistema que deve ser desenvolvido. Na fase de desenvolvimento realiza-se o projeto do sistema, com descrições das principais estruturas de dados e algoritmos, sua codificação, com a implementação do projeto em termos de uma linguagem de programação, e testes dos programas desenvolvidos. Na fase de manutenção são realizadas

Menemónica		Descrição		Flag	CLK	Notas
		Transferência de dados				
MOVLW	k	Mova literal para W	k → W		1	
MOVWF	f	Mova W para f	W → f		1	
MOVF	f, d	Mova f	f → d	Z	1	1, 2
CLRW	-	Clear W (limpar W)	0 → W	Z	1	
CLRF	f	Clear f (limpar f)	0 → f	Z	1	2
SWAPF	f, d	Swap nibbles in f (trocar)	f (7:4),(3:0)→f (3:0),(7:4)		1	1, 2
		Lógicas e Aritméticas				
ADDLW	k	Adicionar literal a W	W+k → W	C, DC, Z	1	
ADDWF	f, d	Adicionar W a f	W+f → d	C, DC, Z	1	1, 2
SUBLW	k	Subtrair W de literal	k-W → W	C, DC, Z	1	
SUBWF	f, d	Subtrair W de f	f-W → d	C, DC, Z	1	1, 2
ANDLW	k	AND literal com W	W .AND. k → W	Z	1	
ANDWF	f, d	AND W com f	W .AND. f → d	Z	1	1, 2
IORLW	k	Inclusivo OR de literal com W	W .OR. k → W	Z	1	
IORWF	f, d	Inclusivo OR de W com f	W .OR. f → d	Z	1	1, 2
XORWF	f, d	Exclusivo OR de W com f	W . XOR. f → d	Z	1	1, 2
XORLW	k	Exclusivo OR de literal com W	W .XOR. k → W	Z	1	
INCF	f, d	Incrementar f	f+1 → f	Z	1	1, 2
DECF	f, d	Decrementar f	f-1 → f	Z	1	1, 2
RLF	f, d	Rode f p/ esquerda com o carry	C ← 7 6 5 4 3 2 1 0 ←	C	1	1, 2
RRF	f, d	Rode f p/ a direita com o carry	→ 7 6 5 4 3 2 1 0 → C	C	1	1, 2
COMF	f, d	Complementar f	$\overline{f}$ → d	Z	1	1, 2
		Operações sobre bits				
BCF	f, b	Bit Clear f (bit de f a '0')	0 → f (b)		1	1, 2
BSF	f, b	Bit Set f (bit de f a '1')	1 → f (b)		1	1, 2
		Direccionamento do programa				
BTFSC	f, b	Bit Test f, Salte se Clear ('0')	salte se f(b)=0		1(2)	3
BTFSS	f, b	Bit Test f, Salte se Set ('1')	salte se f(b)=1		1(2)	3
DECFSZ	f, d	Decremente f, salte se der 0	f-1 → d, salte se der 0		1(2)	1, 2, 3
INCFSZ	f, d	Incremente f, salte se der 0	f+1 → d, salte se der 0		1(2)	1, 2, 3
GOTO	k	Go to address (Ir p/ endereço)	k → PC		2	
CALL	k	Chamar subrotina	PC → TOS, k → PC		2	
RETURN	-	Retorno de subrotina	TOS → PC		2	
RETLW	k	Retorno com literal em W	k → W, TOS → PC		2	
RETFIE	-	Retorno de interrupção	TOS → PC, 1 → GIE		2	
		Outras instruções				
NOP	-	Nenhuma operação			1	
CLRWDT	-	Temporizador do Watchdog=0	0 → WDT, 1 → TO, 1 → PD	$\overline{TO}$, $\overline{PD}$	1	
SLEEP	-	Entrar no modo 'sleep'	0 → WDT, 1 → TO, 0 → PD	$\overline{TO}$, $\overline{PD}$	1	

Figura 1.1: Conjunto de instruções Assembly da família PIC16Cxx de microcontroladores.

modificações decorrentes da correção de erros e atualizações do programa.

Uma vez estabelecida a funcionalidade do programa que se deseja implementar na fase de definição, a fase de desenvolvimento propriamente dita pode ser iniciada. A fase de desenvolvimento costuma tomar a maior parte do ciclo de vida de criação de um programa. Nesta fase são tomadas decisões que podem afetar sensivelmente o custo e a qualidade do software desenvolvido. Esta fase pode ser dividida em três etapas principais, que são projeto, codificação e teste.

Dentre as diversas **técnicas de engenharia de software** presentes atualmente a que vêm se destacando fortemente por facilitar abstrações de objetos, é a técnica UML (*Unified Modular Language*) para especificar todo um projeto de software, incluindo relacionamento lógico entre os dados do programa, interface com o usuário e arquivo, fluxo de dados no tempo. Para os mais interessados, ver Apêndice A.

1.3 Projeto de Software

O projeto de software pode ser subdividido em dois grandes passos, projeto preliminar e projeto detalhado. O projeto preliminar preocupa-se em transformar os requisitos especificados na fase de análise em arquiteturas de dados e de software. O projeto detalhado refina estas representações de arquitetura em

```
;*****Escolher e configurar um microcontrolador*****

      PROCESSOR 16f84
      #include "p16f84.inc"

         __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****     Declarar variáveis     *****

         Cblock 0x0C                ; Início da RAM
         WCYCLE                     ; Pertence à macro  'WAITX'
         PRESCwait
         endc

;***** Estrutura da memória de programa *****

         ORG     0x00               ; Vector de reset
         goto    Main

         ORG     0x04               ; Vector de interrupção
         goto    Main               ; Não há rotina de interrupção

         #include "bank.inc"        ; Ficheiros auxiliares
         #include "button.inc"
         #include "wait.inc"

Main                                ; Início do programa
         BANK1
         movlw   0xff               ; Iniciação do Porto A
         movwf   TRISA              ; TRISA <- 0xff
         movlw   0x00               ; Iniciação do Porto B
         movwf   TRISB              ; TRISB <- 0x00
         BANK0

         clrf     PORTB             ; Porto B <- 0x00

Loop
         Button 0, PORTA, 3 , .100, On    ; Tecla 1
         Button 0, PORTA, 2 , .100, Off   ; Tecla 2
         goto     Loop

On
         bsf      PORTB,7           ; Acender LED
         return

Off
         bcf      PORTB,7           ; Apagar LED
         return

   End            ; Fim de programa
```

Figura 1.2: Exemplo de código Assembly para PIC16C84.

estruturas de dados detalhadas e em representações algorítmicas do programa.

Uma das estratégias de projeto mais utilizadas é o **desenvolvimento top-down**. Neste tipo de desenvolvimento, trabalha-se com o conceito de refinamento de descrições do programa em distintos níveis de abstração. O conceito de abstração está relacionado com esconder informação sobre os detalhes. No nível mais alto de abstração, praticamente nenhuma informação é detalhada sobre como uma dada tarefa será implementada – simplesmente descreve-se qual é a tarefa. Em etapas sucessivas de refinamento, o projetista de software vai elaborando sobre a descrição da etapa anterior, fornecendo cada vez mais detalhes sobre como realizar a tarefa.

O desenvolvimento top-down estabelece o processo de passagem de um problema a uma estrutura de software para sua solução (Fig. 1.3), resultando em uma estrutura que representa a organização dos distintos componentes (ou módulos) do programa. Observe que a solução obtida pode não ser única: dependendo de como o projeto é desenvolvido e das decisões tomadas, distintas estruturas podem resultar.

Outro aspecto tão importante quanto a estrutura de software é a estrutura de dados, que é uma representação do relacionamento lógico entre os elementos de dados individuais.

Uma vez estabelecidas as estruturas de software e de dados do programa, o detalhamento do projeto

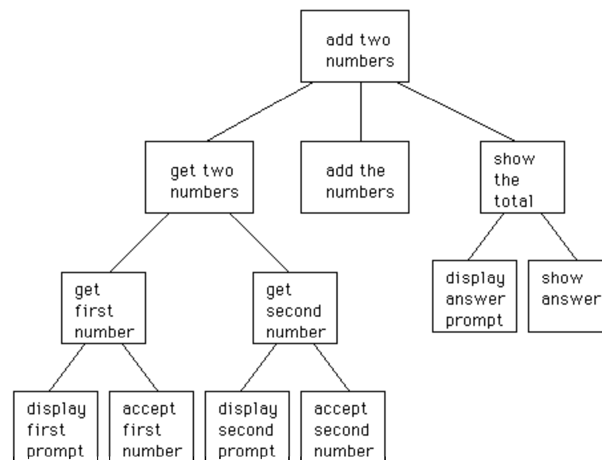


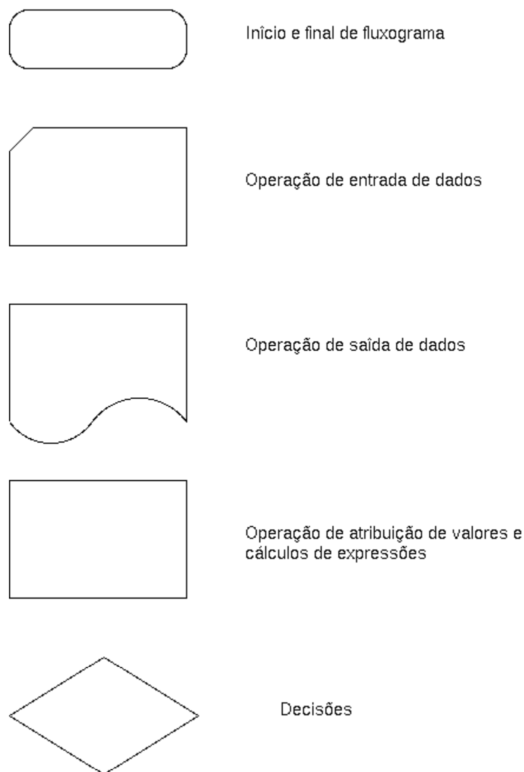
Figura 1.3: Exemplo de Metodologia top-down.

pode prosseguir com o projeto procedimental, onde são definidos os detalhes dos algoritmos que serão utilizados para implementar o programa. Um **algoritmo** é uma solução passo-a-passo para a resolução do problema especificado que sempre atinge um ponto final. Em princípio, algoritmos poderiam ser descritos usando linguagem natural (português, por exemplo). Entretanto, o uso da linguagem natural para descrever algoritmos geralmente leva a ambigüidades, de modo que se utilizam normalmente linguagens mais restritas para a descrição dos passos de um algoritmo. Embora não haja uma representação única ou universalmente aceita para a representação de algoritmos, dois dos mecanismos de representação mais utilizados são o **fluxograma** (ver Fig. 1.4 e a descrição em **pseudo-linguagens** (por exemplo: Portugol[1]).

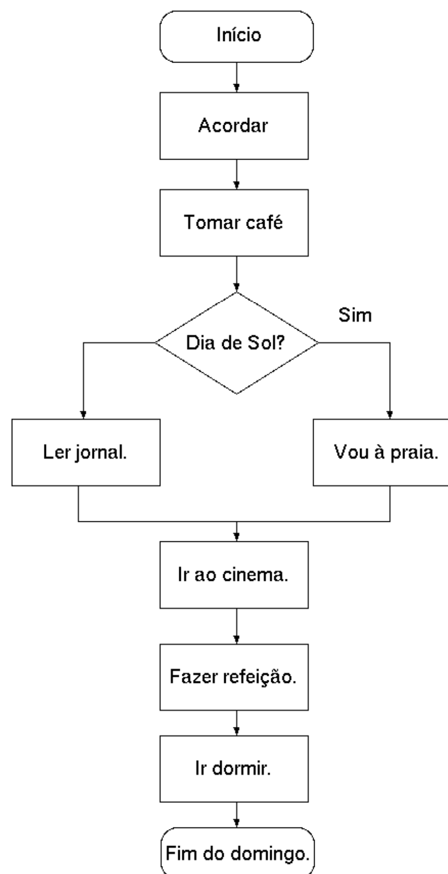
Um **fluxograma** é uma representação gráfica do fluxo de controle de um algoritmo, denotado por setas indicando a seqüência de tarefas (representadas por retângulos) e pontos de tomada de decisão (representados por losangos). A descrição em pseudo-linguagem combina descrições em linguagem natural com as construções de controle de execução usualmente presentes em linguagens de programação.



[1]"Portugol"é derivado da aglutinação de Português + Al-gol. Algol é o nome de uma linguagem de programação estruturada usada no final da década de 50.



a) Fluxogramas típicos.



b) Exemplo de fluxograma.

Figura 1.4: Fluxogramas.

1.4 Algoritmos

Para resolver um problema no computador é necessário que seja primeiramente encontrada uma maneira de descrever este problema de uma forma clara e precisa. É preciso que encontremos uma seqüência de passos que permitam que o problema possa ser resolvido de maneira automática e repetitiva. Esta seqüência de passos é chamada de algoritmo. Um exemplo simples e prosaico de como um problema pode ser resolvido se fornecermos uma seqüência de passos que mostrem a solução é uma receita de bolo.

Algoritmos são uma seqüencia de instruções que se seguidas realizam determinada tarefa. Pode ser também um conjunto finito de regras que fornece uma seqüencia de operações para resolver um problema específico. Nesta seção trataremos não apenas de algoritmos, mas também das **estruturas de dados** que eles utilizam. Este um é dos principais pontos quando lidamos com programação de computadores.

No seu livro "Fundamental Algorithms", vol. 1, Donald Knuth apresenta uma versão para a origem desta palavra. Ela seria derivada do nome de um famoso matemático persa chamado Abu Ja´far Maomé ibn Mûsâ al-Khowârism (825) que traduzido literalmente quer dizer Pai de Ja´far, Maomé, filho de Moisés, de Khowârizm. Khowârizm é hoje a cidade de Khiva, na ex União Soviética. Este autor escreveu um livro chamado Kitab al jabr w´al-muqabala (Regras de Restauração e Redução). O título do livro deu origem também a palavra "Álgebra".

Assim um algoritmo é uma solução passo-a-passo para a resolução do problema especificado que sempre atinge um ponto final.

Exemplos de algoritmos:
Instruções para Receitas de Bolo de Cenoura:

Receita 1 (*Solução 1*):

Ingredientes: 1) 2 xícaras (chá) de cenoura picada; 2) 2 ovos; 3) 1 xícara (chá) de óleo de milho; 4) 1 e 1/2 xícara (chá) de açúcar; 5) 1 xícara (chá) de maizena; 6) 1 xícara (chá) de farinha de trigo; 7) 1 colher (sopa) de fermento em pó. **Cobertura:** 1) 4 colheres (sopa) de leite morno; 2) 4 colheres (sopa) de chocolate em pó; 3) 6 colheres (sopa) de açúcar.	Entrada(s)...
Modo de Preparo: 1) Bata a cenoura com os ovos no liqüidificador; 2) Passe para uma tigela e acrescente os demais ingredientes, misturando bem; 3) Leve ao forno médio, numa assadeira (média) untada e enfarinhada, cerca de 25 minutos; 4) Misture todos os ingredientes do glacê; 5) Retire o bolo do forno e cubra com o glacê; 6) Deixe esfriar e corte em quadrados.	Processamento...
Rendimento: 1 bolo.	Saída...



Receita 2 (*Solução 2*):

Ingredientes: 1) 1 tablete de margarina; 2) 2 ovos; 3) 14 colheres rasas (sopa) de açúcar; 4) 14 colheres cheias (sopa) de farinha de trigo; 5) 2 copos de leite; 6) 1 lata de leite condensado; 7) 2 colheres (sopa) de fermento; 8) 3 colheres (sopa) de chocolate em pó; 9) chocolate granulado; 10) 1 vidro de leite de coco; 11) coco ralado.	Entrada(s)...
Modo de Fazer: 1) Bata no liquidificador: a margarina, os ovos, a farinha de trigo, o açúcar, o chocolate em pó, baunilha ou o leite de coco. 2) Numa tigela coloque a farinha de trigo. 3) Entorne aos poucos, o creme batido, 1 copo de leite e o fermento. 4) Leve ao forno, em assadeira grande untada e enfarinhada. 5) Depois de assado, ainda quente, perfure com um garfo e, coloque leite condensado e, ou coco ralado, ou chocolate granulado.	Processamento...
Rendimento: 1 bolo.	Saída...

Um algoritmo opera sobre um conjunto de **entradas** (no caso do bolo, farinha ovos, fermento, etc.) de modo a gerar uma **saída** que seja útil (ou agradável) para o usuário (o bolo pronto).

Um algoritmo tem cinco características importantes:

Finitude: Um algoritmo deve sempre terminar após um número finito de passos.

Definição: Cada passo de um algoritmo deve ser precisamente definido. As ações devem ser definidas rigorosamente e sem ambiguidades.

Entradas: Um algoritmo deve ter zero ou mais entradas, isto é quantidades que são lhe são fornecidas antes do algoritmo iniciar.

Saídas: Um algoritmo deve ter uma ou mais saídas, isto é quantidades que tem uma relação específica com as entradas.

Efetividade: Um algoritmo deve ser efetivo. Isto significa que todas as operações devem ser suficientemente básicas de modo que possam ser em princípio executadas com precisão em um tempo finito por um humano usando papel e lápis.

É claro que todos nós sabemos construir algoritmos. Se isto não fosse verdade, não conseguiríamos sair de casa pela manhã, ir ao trabalho, decidir qual o melhor caminho para chegar a um lugar, voltar para casa, etc. Para que tudo isto seja feito é necessário uma série de entradas do tipo: a que hora acordar, que hora sair de casa, qual o melhor meio de transporte, etc.

Um fator importante é pode haver mais de um algoritmo para resolver um problema. Por exemplo, para ir de casa até o trabalho, posso escolher diversos meios de transportes em função do preço, conforto, rapidez, etc. A escolha será feita em função do critério que melhor se adequar as nossas necessidades (tempo de processamento, armazenamento).



"Ao verificar que um dado programa está muito lento, uma pessoa prática pede uma máquina mais rápida ao seu chefe. Mas o ganho potencial que uma máquina mais rápida pode proporcionar é tipicamente limitado por um fator de 10, por razões técnicas ou econômicas. Para obter um ganho maior, é preciso buscar ***melhores algoritmos****. Um bom algoritmo, mesmo rodando em uma máquina lenta, sempre acaba derrotando (para instâncias grandes do problema) um algoritmo pior rodando em uma máquina rápida. Sempre."*
S. S. Skiena, *The Algorithm Design Manual*

Por analogia, um pacote de *Software* pronto = pacote de bolo industrializado.



1.5 Programação Estruturada e Fluxogramas

A programação estruturada estabelece uma disciplina de desenvolvimento de algoritmos que facilita a compreensão de programas através do número restrito de mecanismos de controle da execução de programas. Qualquer algoritmo, independentemente da área de aplicação, de sua complexidade e da linguagem de programação na qual será codificado, pode ser descrito através destes mecanismos básicos.

O princípio básico de programação estruturada é que um programa é composto por blocos elementares de código que se interligam através de três blocos básicos, que são **entrada de dados**, **processamento** e **saída de dados** – ver figura 1.5.

Cada uma destas construções tem um **ponto de início** (o topo do bloco) e um **ponto de término** (o fim do bloco) de execução – ver figura 1.6.

A figura 1.7 mostra os fluxogramas que utilizaremos nesta disciplina.

O bloco "**processamento**" implementa os passos de processamento necessários para descrever qualquer programa. Por exemplo, um segmento de programa da forma "faça primeiro a Tarefa_a e depois a Tarefa_b"

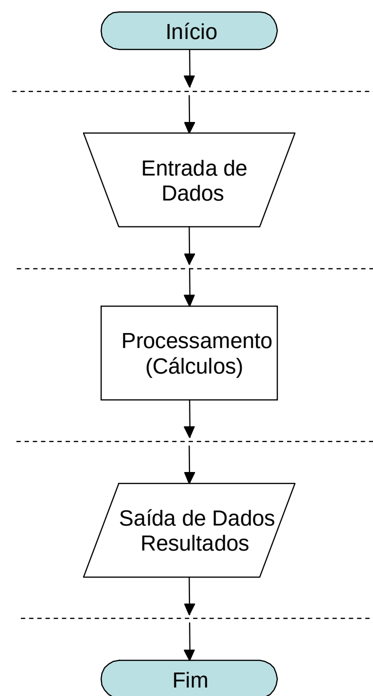


Figura 1.5: Principais blocos de uma programação estruturada.

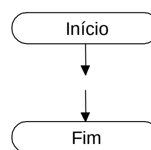


Figura 1.6: Início e fim de cada bloco.

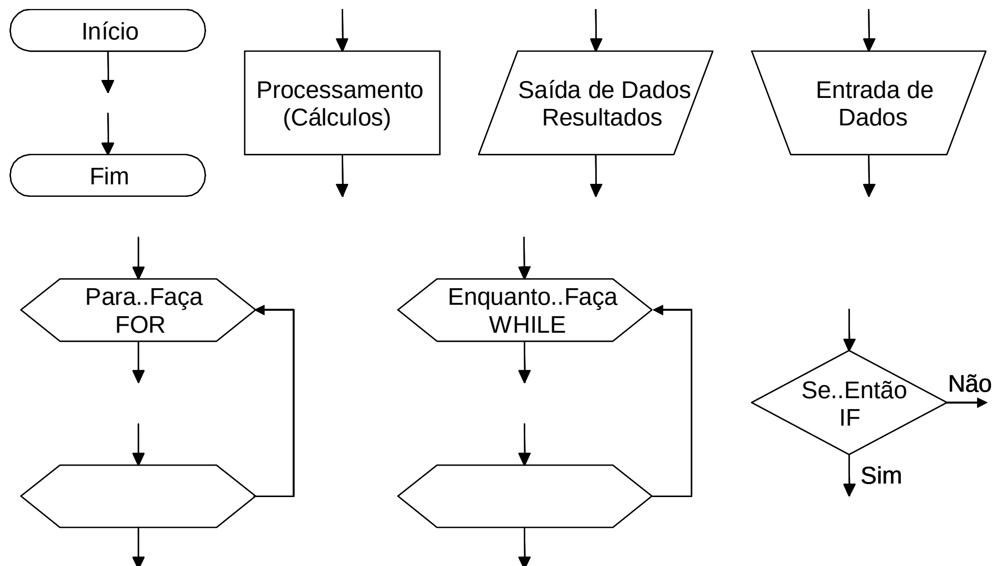


Figura 1.7: Fluxogramas utilizados neste curso.

seria representado por uma seqüência de dois retângulos (Fig. 1.8a). A mesma construção em pseudo-linguagem seria denotada pela expressão das duas tarefas, uma após a outra (Fig. 1.8b).

Como no caso do exemplo da figura 1.3 uma forma de documentar na forma de um fluxograma um programa que soma dois números poderia ficar como mostra a figura 1.9. Note pela figura 1.9 que aparecem as "variáveis" de programação: Num_1, Num_2 e Soma. No próximo item entraremos no item que se refere ao que se chama de **variáveis**.

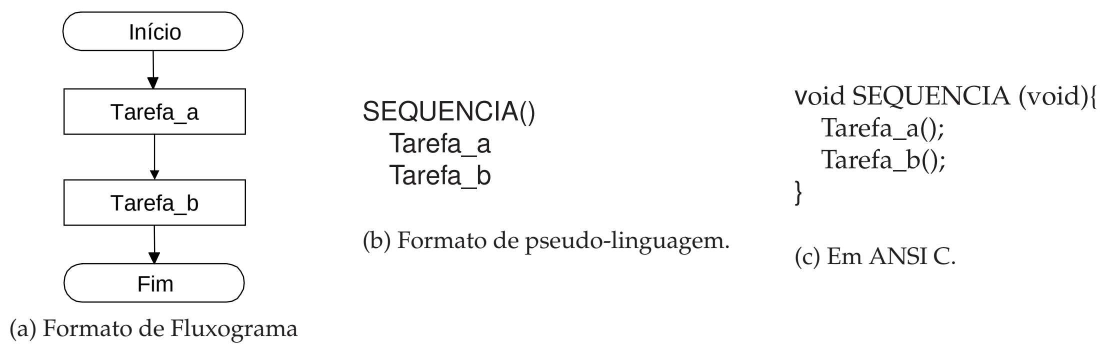


Figura 1.8: Exemplo de blocos de processamento

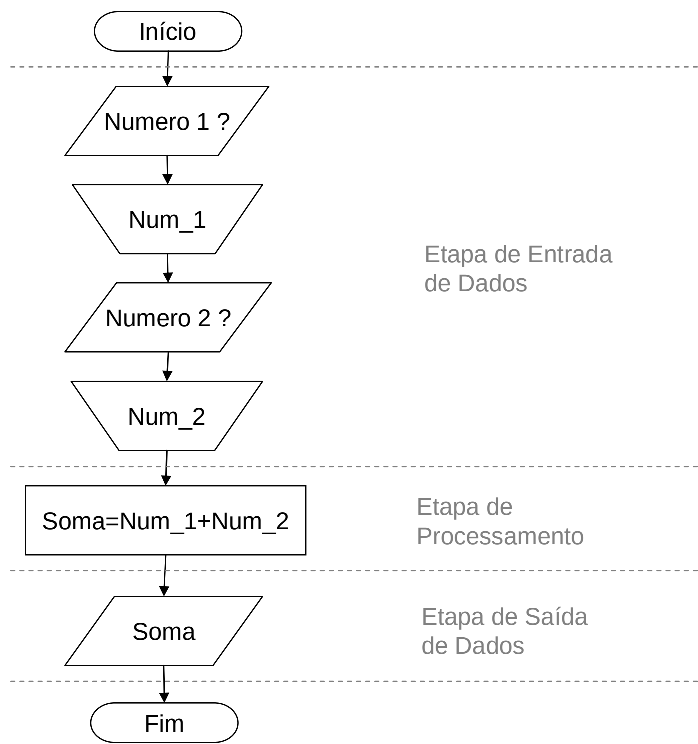


Figura 1.9: Exemplo de fluxograma para soma de 2 números.

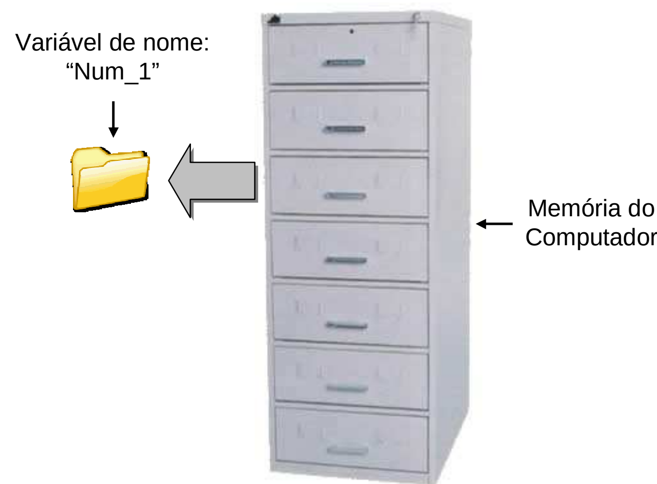


Figura 1.10: Analogia entre variáveis e pastas de arquivo suspenso.

1.6 Variáveis

Um computador necessita organizar os dados que manipula na forma de **variáveis**. Como o próprio nome indica, seu conteúdo é "variável", isto é, serve para armazenar valores que podem ser modificados durante a execução de um programa.

A figura 1.10 mostra uma analogia para o que seriam "variáveis" comparando a forma como o computador manipula dados e a forma como nós seres humanos podemos organizar informação escrita (documentos).

Note que a memória do computador poderia ser comparada à um arquivo de aço. Uma pasta suspensa dentro deste arquivo seria o equivalente ao que se chama de variável. Assim como podemos etiquetar (colocar um nome) na pasta suspensa, o usuário é quem define o nome que prefere para suas variáveis. Note que cada variável guarda um certo tipo de dado (ou informação). Quando dentro de um programa o usuário faz referência à uma variável, o computador automaticamente sabe onde está localizada esta variável dentro da sua memória (o que de certa forma explica porque um computador pode ser mais rápido para recuperar dados que um ser humano buscando um certo documento dentro de um arquivo de aço).

Note que quanto maior o arquivo de aço, mais documentos podemos estocar dentro do mesmo. Da forma, é assim com o computador, quanto mais **memória RAM**, maior a quantidade de dados com a qual ele pode lidar instantaneamente. No caso de não haver memória RAM suficiente para lidar com os dados em questão (uma folha de pagamentos por exemplo), eles podem ser gravados numa memória secundária (disco rígido, CD, DVD, etc), na forma do que chamamos de "arquivos". Neste último caso, o sistema operacional (S.O.) que está sendo rodado pelo computador é que vai definir a maior facilidade ou não para acessar estes arquivos e é o S.O. que se responsabiliza por localizar os arquivos com os dados desejados pelo computador.

A cada variável corresponde uma posição de memória, cujo conteúdo pode variar ao longo do tempo durante a execução do programa.

Note que assim com num arquivo de pastas suspensas, vários tipos de informações podem ser guardadas: folhas simples, folhas vazias, fotografias, fitas cassetes, CD's, DVD's, etc. Da mesma forma o computador pode lidar com diferentes tipos de dados ou variáveis. Note que depende da capacidade de **abstração** e experiência do usuário, a definição das variáveis que serão necessárias para executar um certo programa.

Note ainda que é o programador que vai definir se existe ou não uma **lógica de relacionamento entre diferentes dados** e como se dá este relacionamento. No exemplo anterior (da Fig. 1.9), é a variável "Soma" que vai receber o resultado da soma do conteúdo de outras duas variáveis: "Num_1" e "Num_2":

$$\text{Soma} \leftarrow \text{Num_1} + \text{Num_2}$$

Note também que quem **atribui** valor para as variáveis "Num_1" e "Num_2" é o usuário via entrada de dados pelo teclado (indicado na figura 1.9 pelo bloco referente à "Entrada de Dados" – voltar a figura 1.7 se for o caso).

1.6.1 EXERCÍCIOS

1) Como seria um simples fluxograma para fazer aparecer na tela uma mensagem como "Alô Mundo!"?

2) Esboce na forma de um fluxograma o algoritmo correspondente ao cálculo da média de 3 valores informados pelo usuário.

3) Monte um fluxograma correspondente a um algoritmo capaz de calcular o valor final atingido por uma mercadoria, dado que o usuário informa o valor inicial da mercadoria e a porcentagem de desconto dada pela loja.

4) Desenhe como seria o fluxograma correspondente à um programa capaz de calcular as raízes para uma equação do 2º grau: $ax^2 + bx + c = 0$, onde o usuário entra com os coeficientes da equação: a, b e c.

Note: para todos os fluxogramas acima, cada aluno deve abstrair que variáveis serão necessárias para que o programa funcione, a seqüencia (correta) de instruções (blocos de fluxograma) e quais seriam os blocos necessárias para garantir a efetividade do algoritmo. Como um primeiro auxílio para esta etapa do processo, o aluno pode montar uma **tabela chamada "IPO"**[2] que o auxilie a estruturar a forma como deve ser o programa – ver figura 1.11.

Entrada	Processamento	Saída

Figura 1.11: Diagrama IPO (vazio).

A figura 1.12 mostra como ficaria este digrama para o item (1) dos EXERCÍCIOS.

Entrada	Processamento	Saída
(Nenhuma)	**Imprima "Alô Mundo"**	**Alô Mundo**

Figura 1.12: Diagrama IPO para o item (1) dos EXERCÍCIOS.

1.6.2 Tipos de Dados

Tipos de Dados Básicos

Como podemos perceber pelo item anterior, o computador exige que o usuário defina uma variável relacionada cada dado à ser manipulado. Ocorre que para o computador existem 4 tipos básicos de dados: *int* (inteiro), *float* (real), *double* (real de dupla precisão) e *char* (carácter) que são tratados pelo mesmo, de diferentes formas. A tabela 1.1 mostra os diferentes tipos de dados básicos disponíveis no ANSI C.

Tipo de Dado	Bytes Exigidos	Faixa de Variação
char	1	0 à 255
int	2	-32.768 à 32.767
float	4	$3.4 \times 10^{\pm 38}$ (7 dígitos)
double	8	$1.7 \times 10^{\pm 308}$ (15 dígitos)

Tabela 1.1: Tipos de dados básicos do ANSI C.

O tipo de dados *int* serve para guardar números inteiros, positivos e negativos.

Variáveis inteiras podem ser qualificadas na sua declaração como *short* ou *long* e *unsigned*. Um tipo *unsigned int* indica que a variável apenas armazenará valores positivos. O padrão é que variáveis inteiras utilizem a representação em complemento de dois (binariamente falando), para valores positivos e negativos.

[2] "IPO" = *Input Process Output diagrams* = ou Diagrama Entrada Processamento Saída.

Tipos de Dados Extendidos

Os tipos de dados básicos foram extendidos através de "**modificadores**" de forma a extender e tornar mais específicas certos tipos de dados básicos – ver tabela 1.2. Os modificadores *short* e *long* modificam o espaço reservado para o armazenamento da variável. Um tipo *short int* indica que (caso seja possível) o compilador deverá usar um número menor de bytes para representar o valor numérico – usualmente, dois bytes são alocados para este tipo. Uma variável do tipo *long int* indica que a representação mais longa de um inteiro deve ser utilizada, sendo que usualmente quatro bytes são reservados para variáveis deste tipo.

Tipo de Dado	Bytes Exigidos	Faixa de Variação
char	1	0 à 255
signed char	1	-128 à 127
unsigned char	1	0 à 255
short	2	-32.768 à 32.767
unsigned short	2	0 à 65.535
int	2	-32.768 à 32.767
unsigned int	2	0 à 65.535
long	4	-2.147.483.648 à 2.147.483.647
unsigned long	4	0 à 4.294.967.295
float	4	$3.4 \times 10^{\pm 38}$ (7 dígitos)
double	8	$1.7 \times 10^{\pm 308}$ (15 dígitos)
long double	10	$1.2 \times 10^{\pm 4932}$ (19 dígitos)

Tabela 1.2: Tipos de dados extendidos do ANSI C.

O tipo de dados *char* serve para guardar um caracter, por exemplo a letra 'Z'.

Reparem que não importa o tipo de dados, todos eles são codificados com seqüencias de 0's e 1's para o computador (seqüencias de bits). Para representar um número inteiro sem sinal de valor igual à 75 seriam necessários: 7 bits ($2^6 = 64$ e $2^7 = 128$, $75_{(10)} = 01001011_{(2)}$).

Já para representar um caracter são usadas tabelas padrões de conversão de um determinado caracter para a correspondente seqüencia binária – por exemplo, a tradicional **tabela ASCII** (ver figura 1.13).

```
     0         1         2         3         4         5
     01234567890123456789012345678901234567890123456789
  0                                   !"#$%&'()*+,-./01
 50  23456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
100  defghijklmnopqrstuvwxyz{|}~⌂ÇüéâäàåçêëèïîìÄÅÉæÆôöò
150  ûùÿÖÜø£Ø×ƒáíóúñÑªº¿®¬½¼¡«»░▒▓│┤ÁÂÀ©╣║╗╝¢¥┐└┴┬├─┼ãÃ
200  ╚╔╩╦╠═╬¤ðÐÊËÈıÍÎÏ┘┌█▄¦Ì▀ÓßÔÒõÕµþÞÚÛÙýÝ¯´-±‗¾¶§÷¸°¨
250  ·¹³²■
```

Figura 1.13: Tabela de códigos ASCII

Assim por exemplo, o código ASCII para a letra 'A' é 65 ou $65_{(10)} = 01000001_{(2)}$. Note que cada carácter gasta exatamente 1 byte (ou 8 bits). E mais, perceba que o **caracter** '7', binariamente não têm nenhuma relação com o número em decimal 7:

$$\begin{array}{rcll} \text{O número 7} & \rightarrow & 0000 \quad 0111\ _{(2)} & \\ \text{O \textbf{caracter} '7'} & \rightarrow & 0011 \quad 0111\ _{(2)} & = 55\ _{(10)} \end{array}$$

Os tipos de dados *float* e *double* são usados para guardar números reais (número com ponto flutuante, ou com casas decimais). A diferença entre eles é a precisão. O *float* tem uma precisão de 6 casas decimais e o *double* tem uma precisão de 10 casas decimais.

Valores com representação em ponto flutuante (números reais) são representados em C através do uso do ponto decimal, como em `1.5` para representar o valor um e meio. A notação exponencial também pode ser usada, como em `1.2345e-6` ($= 0,0000012345$) ou em `0.12E3` ($= 120$).

Os números reais (em ponto flutuante) também são codificados numa seqüencia de 0s e 1s na memória do computador só que de uma forma diferente da utilizada para números inteiros. Para armazenar números reais o computador representa o número na forma de uma mantissa e um expoente, gastando uma quantidade diferente de bytes para os mesmos. Por exemplo, o número real $6,574$ pode estar armazenado na memória do computador no formato: mantissa igual à 6574 (2 bytes) e expoente igual à -3 (1 byte), ou seja, $6,574 = 6574 \times 10^{-3}$.

Assim, nos computadores os números reais não são um conjunto contínuo e infinito como na Matemática. Isso só seria possível se a memória dos computadores também fosse infinita (quantidade infinita de bytes disponíveis). **Na prática, isso pode não se transformar num problema maior porque grande parte dos problemas podem resolver-se limitando os cálculos à precisão desejada**. A diferença entre as variáveis do tipo *float* e *double* é que as variáveis do tipo *double* têm mais precisão, isto é, são representadas internamente no computador usando mais bits.

Estas dimensões de variáveis denotam apenas uma situação usual definida por boa parte dos compiladores, sendo que não há nenhuma garantia quanto a isto. A única coisa que se pode afirmar com relação à dimensão de inteiros em C é que uma variável do tipo *short int* não terá um número maior de bits em sua representação do que uma variável do tipo *long int*.

Algumas linguagens possuem ainda outras formas mais sofisticadas de representar dados, como no caso de **Programação Orientada à Objetos** (POO), onde um *objeto* nada mais é do tipo específico de abstração de dados, especialmente definida para a aplicação em vista. Por isto, programar orientado à objetos exige uma nova filosofia de programação não suportada nesta disciplina. Mas à título de curiosidade, para o MATLAB®, qualquer tipo de dado sendo processado pelo mesmo é na realidade um objeto e isto fica mais fácil de compreender quando nos lembramos de que o MATLAB® está preparado para lidar com números complexos que possui parte real e parte imaginária. Note ainda que nas linguagens orientadas à objeto, no momento da definição de um objeto, por exemplo, no momento da definição da *classe* ***complex*** por exemplo, também podem ser codificados a forma como o sistema deve executar operações de adição, soma, subtração, divisão, exponenciação, etc – é desta forma que o MATLAB® lida de modo "transparente" com números complexos.

EXERCÍCIOS: Volte à lista de EXERCÍCIOS da página 12 e identifique o tipo de variável adequado para cada um dos dados definidos em cada exercício.

1.6.3 Nomes de Variáveis (ou "Identificadores")

Toda variável é identificada por um nome ou **identificador**. Identificadores são nomes escolhidos para representar dados. Ocorre porém que certas **regras** devem ser levadas em conta na escolha de um identificador para uma variável:

Comprimento limitado: A princípio, o nome adotado para uma variável poderia ter qualquer comprimento, mas na prática, seu valor varia conforme a linguagem de programação sendo utilizada e seu compilador. O ANSI C distingue apenas os primeiros **31** caracteres utilizados para identificar uma variável.
Por exemplo, as variáveis abaixo são idênticas para o ANSI C:
`1234567890123456789012345678901` ← Quantidade de caracteres
`Minha_Mae_disse_para_eu_escolher_este_nome_aqui = Minha_Mae_disse_para_eu_escolhe,`
`Testa_Sensor_Esquerda_Video_Cassete = Testa_Sensor_Esquerda_Video_Cas,`
Exemplo à seguir: no limite dos 31 caracteres:
`TestaSensorEsquerdaVideoCassete`.
Outros exemplos: Sensor1, NomeArquivo, Nome_Arquivo, NomArq, NA.



Note: Exagerar no comprimento do nome utilizado para identificar uma variável pode tornar a codificação de um programa extremamente difícil: você já pensou em digitar várias vezes durante o programa a referência para a variável "TestaSensorEsquerdaVideoCassete" ? Ao mesmo tempo, letras simples, ou nomes muito curtos ou mnemônicos dificultam a compreensão do programa. Assim, o melhor é ser razoável, usar o bom senso e escolher nomes que agreguem um significado até intuitivo para cada variável (isto é, o próprio nome escolhido para a variável, já indica a sua função).

Primeiro caracter: uma letra – obrigatoriamente.
Exemplos Errados: 10, 10E, 102Elefante ← além de não agregarem nenhum sentido à variável, não são nomes válidos para uma variável.
Exemplos "corretos": E10, Elefante102.

Após a primeira letra: letras, dígitos ou sublinhado: Pode-se usar letra, números ou o caracter sublinhado ("_").
Por exemplo: Sensor_1, Nome_Aluno.

Letras maiúsculas ≠ letras minúsculas: A maioria das linguagens de programação faz distinção entre letras minúsculas e maiúsculas.



Por exemplo: No ANSI C:
Nome_Arq ≠ Nome_arq ← Note que mudou apenas uma letra!
INDEX ≠ index ≠ InDeX.
Desta forma, o programador deve ser cuidadoso no sentido de sempre ter que lembrar-se a **sintaxe** (grafia) correta usada para identificar suas variáveis.

Não podem haver identificadores repetidos: Isto é, cada variável deve possuir um nome diferente de todos os outros.

Não pode ser uma palavra reservada: da linguagem de programação em questão.
Por exemplo: C, possui 32 palavras reservadas; nomes que não podem ser adotados como identificadores para variáveis – ver tabela 1.3 à seguir. Note que são instruções da linguagem C e estão todas em letras minúsculas.



auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Tabela 1.3: Tabela de palavras reservadas do ANSI C.

1.7 Comandos de Atribuição de Variáveis

Atribuir valores à uma variável significa armazenar valores dentro da mesma. A tabela 1.4 à seguir demonstra como atribuir valores à variáveis.

a	←	3
i	←	i + 1
delta	←	$b^2 - 4 \cdot a \cdot c$
x	←	"A"

(a) Em PORTUGOL

```
      a  =  3;
      i  =  i + 1;
  delta  =  b*b - 4*a*c;
 char x  =  'A';
```

(b) Em ANSI C.

Tabela 1.4: Comandos de atribuição de variáveis.

1.7.1 Expressões Aritméticas

Expressões aritméticas são aquelas que apresentam como resultado um valor numérico que pode ser um número inteiro ou real, dependendo dos operandos e operadores. Os **operadores aritméticos** disponíveis são mostrados na tabela 1.5 à seguir.

Operador	Descrição	Hierarquia da Operação
$+$	Soma	$3^{\underline{a}}$
$-$	Subtração	$3^{\underline{a}}$
$*$	Multiplicação	$2^{\underline{a}}$
/	Divisão	$2^{\underline{a}}$
(	(	$1^{\underline{a}}$
)	)	$1^{\underline{a}}$

Tabela 1.5: Tabela de operadores aritméticos básicos e precedências dos mesmos.

A coluna "**Hierarquia da Operação**", indica a **ordem de precedência** de uma operação aritmética (válido para qualquer linguagem de programação). Isto quer dizer que, numa expressão como a seguinte:

$$x = a \cdot (b + 2 \cdot c)$$

os seguintes termos serão avaliados nesta ordem:

$1^{\underline{o}}$) $(b + 2 \cdot c)$

$2^{\underline{o}}$) $2 \cdot c$

$3^{\underline{o}}$) $b +$ Resultado de: "$2 \cdot c$")

$4^{\underline{o}}$) $a \cdot$ Resultado de: "$(b + 2 \cdot c)$")

isto é, primeiro será avaliada a expressão contida dentro dos parênteses ($(b + 2 \cdot c)$), depois será resolvido primeiro a multiplicação: $2 \cdot c$; obtido o resultado da multiplicação é realizada soma com o termo b e por fim é realizada a multiplicação deste resultado pelo termo a.

Exemplos:

	Em linguagem natural:	Em ANSI C:	Resultado:
1)	$a = 3$	`a=3;`	a=3
2)	$i = a + 3$	`i=a+3;`	i=6
3)	$i = i + 1$	`i=i+1;`	i=7
4)	$b = a + i$	`b=a+i;`	b=10
5)	$b = i + a$	`b=i+a;`	b=10
6)	$a = 2b + i$	`a=2*b+i;`	a=27
7)	$a = 2(b + i)$	`a=2*(b+i);`	a=34
8)	$\Delta = b^2 - 4ai$	`delta= b*b - 4*a*i;`	delta=-852
9)	$b = \frac{a}{i}$	`b=a/i;`	b=4.857
10)	$b = 2 + \frac{a}{i}$	`b=2+a/i;`	b=6.8571
11)	$b = 2 + \frac{a}{i}$	`b=2+(a/i);`	b=6.8571
12)	$c = a + \frac{b}{2}$	`c=a+b/2;`	c=37.4286
13)	$c = \frac{a+b}{2}$	`c=(a+b)/2;`	c=20.4286
14)	$z = a\left[\frac{b}{2} + \left(\frac{c+i}{2}\right)\right]$	`z=a*((b/2)+((c+i)/2));` ou `z=a*(b/2+(c+i)/2);`	z=582.8571

Para o exemplo 14) , note formas descuidadas de se codificar esta equação:

ERRADO: `z=a*b/2+(c+i)/2;` → z=130.2857 ← Calculado: $z = a \cdot \frac{b}{2} + \frac{c+i}{2}$

ERRADO: `z=a*(b/2+c+i/2);` → z=930.1429 ← Calculado: $z = a \cdot \left(\frac{b}{2} + c + \frac{i}{2}\right)$

1.7.2 Funções matemáticas

A tabela 1.6 à seguir relaciona as funções matemáticas mais típicas:

Função	Em ANSI C	Exemplo
$\sqrt{x}$	`sqrt(x)`	`y=sqrt(x);`
x^y	`pow(x,y)`	`a=pow(x,y);`
e^x	`exp(x)`	`y=exp(x);`
$\cos(x)$	`cos(x)`	`x=cos(theta);`
$\sin(x)$	`sin(x)`	`y=sin(theta);`
$\tan(x)$	`tan(x)`	`z=tan(x);`
$\arccos(x)$	`acos(x)`	`theta=acos(a);`
$\arcsin(x)$	`asin(x)`	`theta=asin(b);`
$\arctan(x)$	`theta=atan(x)`	`theta=atan(y/x);`
$\arctan(y/x)$	`theta=atan2(y,x)`	`theta=atan2(y,x);`

Obs$_1$: funções trigonométricas são realizadas no espaço de **radianos** e não de graus.
Obs$_2$: estas funções trabalham e retornam variáveis do tipo *real* – SEMPRE!.
Obs$_3$: estas funções exigem o uso da biblioteca <math.h>.

Tabela 1.6: Funções matemáticas mais comuns.

1.7.3 EXERCÍCIOS

Converta as equações matemáticas abaixo para o formato linear, isto é, compatível com uma linguagem de programação como ANSI C:

1) $x = \frac{a+b}{c}$ →

2) $x = a + \frac{b+c}{2}$ →

3) $x = \frac{a}{2} + \frac{b}{4}$ →

4) $x = \frac{\frac{a}{2} + \frac{b}{4}}{3}$ →

5) $y = a + \frac{b}{2+c}$ →

6) $z = a^2 + 2ab + b^2$ →

7) $z = \frac{(a+b)^2}{2}$ →

8) $x = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3}}}$ →

Capítulo 2

Programação (ou Codificação)

2.1 Programar ou Codificar

Uma linguagem consiste essencialmente de uma seqüência de strings ou símbolos com regras para definir quais seqüências de símbolos são válidas na linguagem, ou seja, qual a sintaxe da linguagem. A interpretação do significado de uma seqüência válida de símbolos corresponde à semântica da linguagem.

Existem meios formais para definir a **sintaxe** de uma linguagem – a definição semântica é um problema bem mais complexo. A sintaxe de linguagens é expressa na forma de uma gramática, que será introduzida na seqüência.

A etapa de codificação traduz a representação do projeto detalhado em termos de uma linguagem de programação. Normalmente são utilizadas linguagens de alto nível, que podem então ser automaticamente traduzidas para a linguagem de máquina pelo processo de compilação. Neste texto, a linguagem C será utilizada nos exemplos e atividades práticas desenvolvidas.

2.2 Declaração de Variáveis

Para declarar uma variável de nome *i* do tipo inteiro em um programa C, a seguinte expressão seria utilizada:

```
int i;
```

Essa declaração reserva na memória um espaço para a variável *i*, suficiente para armazenar a representação binária em complemento de dois (com sinal) do valor associado à variável, que inicialmente é indefinido.

É possível, ao mesmo tempo em que se declara uma variável, proceder a sua **inicialização**, isto é, se atribuir um valor inicial definido para a variável, neste caso, a declaração ficaria assim:

```
int i = 0;
```

Atenção nas Operações misturando números Reais e Inteiros



Atenção: Expressões aritméticas podem manipular operandos de dois tipos: reais e inteiros. Se todos os operandos de uma expressão são do tipo inteiro então a expressão fornece como resultado um número inteiro. Caso pelo menos um dos operandos seja real o resultado será real. Isto pode parecer estranho a princípio, mas este procedimento reflete a forma como as operações são executadas pelos processadores.
Por exemplo, o resultado da operação:
`1/5` é `0`, porque os dois operadores são inteiros.
Caso a expressão tivesse sido escrita como `1.0/5` então o resultado `0.2` seria o correto.

2.3 Breve História sobre a Linguagem C

A linguagem C foi criada por Dennis Ritchie em 1972 no Centro de Pesquisas da Bell Laboratories. Sua primeira utilização importante foi a reescrita do Sistema Operacional UNIX que até então era escrito em Assembly.

Em meados de 1970 o UNIX saiu do laboratório para ser liberado para as universidades. Foi o suficiente para que o sucesso da linguagem atingisse proporções tais que, por volta de 1980, já existiam várias versões de compiladores C oferecidas por várias empresas, não sendo mais restritas apenas ao ambiente UNIX, porém compatíveis com vários sistemas operacionais.

O C é uma linguagem de propósito geral, sendo adequado à programação estruturada. No entanto é mais utilizada para escrever compiladores, analisadores léxicos, bancos de dados, editores de textos, etc...

A linguagem C pertence a uma família de linguagens cujas características são: portabilidade[1], modularidade, compilação separada (*"linkagens"*), recursos de baixo nível, geração de código eficiente, confiabilidade, regularidade, simplicidade e facilidade de uso.

2.4 Visão geral de um programa em C

A geração de um programa executável (arquivo com extensão .exe) à partir do programa fonte (no caso, com extensão .c ou .cpp, dependendo do compilador) obedece a uma seqüencia de operações antes de tornar-se executável. Depois de escrever o módulo fonte em qualquer editor de textos, o programador aciona o compilador. Essa ação desencadeia uma seqüencia de etapas, cada qual traduzindo a codificação do usuário para a forma de linguagem de máquina, que termina com o executável criado pelo "linkador" – conforme demonstra a figura 2.1 à seguir.

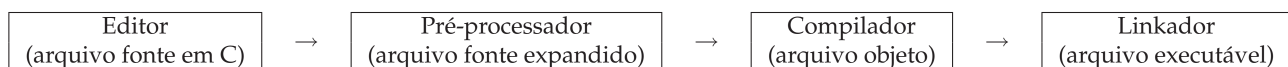


Figura 2.1: Seqüencia de compilação de um programa em C.

2.5 Estrutura Padrão de um programa em C

Estaremos estudando aqui a forma de programar no "**ANSI-C standard**". A linguagem C foi introduzida pelos pesquisadores Kernigan e Ritchie através do livro "**The C Programming Language**". Atualmente já existe a segunda versão deste livro: "**The C Programming Language - Second Edition**" que trata da forma "clássica" de programação em C.

Todos os programas em ANSI C possuem um bloco principal de programação, uma **função** chamada de ***main()***.

As **funções** são as entidades operacionais básicas dos programas em C, que por sua vez são uma união de uma ou várias funções executando cada qual o seu trabalho. Há funções básicas que estão definidas em **bibliotecas em C** (arquivos .h). As funções `printf()` e `scanf()` por exemplo, permitem respectivamente escrever informações na tela e ler dados à partir do teclado – estas funções estão definidas na biblioteca de nome `<stdio.h>`.

Todo programa em ANSI C inicia sua execução chamando a função **`main()`**, sendo obrigatória a sua declaração dentro do programa. Um exemplo do programa mais simples possível de ser realizado em C segue abaixo:

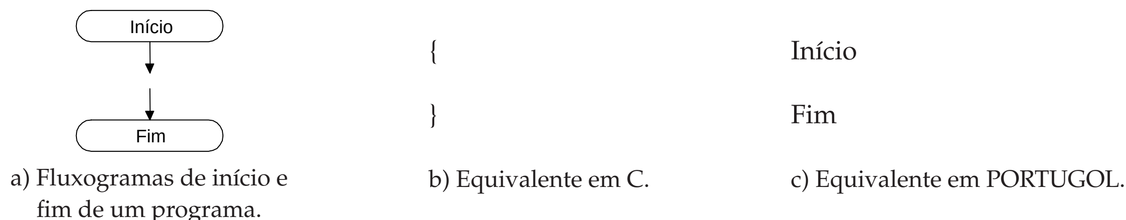
```
main()
{
}
```

[1]há controvérsias

Não existe forma mais simplificada de um programa em C. Infelizmente, este programa não faz nada! Mas serve para nos dar uma pista sobre a estrutura clássica de um programa em C.

Note, a palavra ***main***, é muito importante e aparece sempre e uma única vez em qualquer programa em C. Note que depois da palavra ***main*** existe um par de parênteses que indicam para o compilador que se trata de uma **função** (*function*). Mais tarde (na continuidade desta disciplina) será explicado exatamente o que é uma função e para que servem estes parênteses. Por hora, considere simplesmente incluir SEMPRE este par de parênteses em todos os seus programas.

Note nas linhas 2 e 3 as chaves (caracteres "{" e "}") que definem início e o fim do próprio programa. Equivalente aos blocos de "inicio" e "fim" de um fluxograma, ou em PORTUGOL o que é mostrado na figura 2.2 à seguir.



a) Fluxogramas de início e fim de um programa.

b) Equivalente em C.

c) Equivalente em PORTUGOL.

Figura 2.2: Blocos de início e fim de um programa.

Note que o programa em si, deve vir dentro deste bloco de início e de fim. Note que no exemplo anterior (nosso primeiro programa) não existe nenhuma declaração neste espaço porque justamente este programa não faz nada. Apesar deste programa poder ser compilado e aparentemente o usuário não poder enxergar o código rodando, trata-se de um código perfeitamente válido em C. Eventualmente no momento de compilar este programa poderá ser gerado uma advertência (***warning***), ignore ou modifique o código para:

```
int main()
{
  return 0;
}
```

O código acima deve poder ser compilado em qualquer compilador padrão ANSI C. A única diferença com relação ao primeiro programa é que agora existe uma declaração ***return*** entre as chaves. Como esta própria declaração indica, o programa retorna um valor inteiro para o Sistema Operacional (S0) utilizado para disparar sua execução. No caso, retorna o valor 0 (zero), indicando uma execução bem sucedida. Códigos diferentes de zero podem ser tratados como códigos de erros para o SO (como de fato ocorre em sistemas Unix). Isto ocorre porque na definição original de K&R para a linguagem C, por padrão (por *default*), todas as funções retornam um valor inteiro, independente da vontade do programador. Quando explicitamente não queremos retornar um valor para o SO, podemos tentar escrever o programa como mostrado no primeiro exemplo.

Ou, caso o compilador não aceite compilar este código, uma declarção extra "***void***" (nada em inglês) pode ser utilizada para indicar que efetivamente não desejamos retornar nenhum valor para o SO. Ver exemplo à seguir:

```
void main()
{
}
```

Note que neste último código não aparece mais a declaração "*return*", porque a função *main* não vai retornar nada para o SO. Compiladores modernos aceitam trabalhar com void ou esperam o retorno de um inteiro para a função principal do programa (como esperado para "padrão ANSI C").

2.6 Comandos Básicos de Entrada e Saída de Dados

Esta seção descreve as funções "de entrada" (*input*) e "de saída" (*output*) mais importantes da linguagem C. Todas estão declaradas na biblioteca `<stdio.h>`. Portanto, o seu programa deve ter uma linha contendo a declaração:

```
#include <stdio.h>
```

para poder usar essas funções.

Para que um programa torne-se minimamente funcional é preciso que ele receba dados do meio externo (teclado, mouse, portas de comunicação, drives de disco, etc.) e emita o resultado de seu processamento de volta para o meio externo (monitor, impressora,alto-falante, portas de comunicação, drives de disco, etc.). De outro modo: um programa deve trocar informações com o meio externo. Em C, existem muitas funções pré-definidas que tratam desta troca de informações. São as funções de entrada e saída do C. Nos exemplos mostrados nos capítulos anteriores foram vistas algumas funções de entrada (`scanf()`, `getch()`) e algumas funções de saída (`printf()`). Neste capítulo veremos, em detalhe, estas e outras funções de modo a permitir escrever um programa completo em C.

Mostraremos, nas duas seções iniciais as mais importantes funções de entrada e saída de dados em C: as funções `printf()` e `scanf()`. A partir do estudo destas funções é possível escrever um programa propriamente dito com entrada, processamento e saída de dados.

2.6.1 Comando para Saída de Dados – função: printf()

Biblioteca: `<stdio.h>`

Usos: `printf("`*Mensagem*`");`
`printf ("`*Mensagem %caracteres_controle*`",` *dados*`);`

A função `printf()` (*print formated*) pode imprimir mensagens simples na tela.

Ex_1:

```
#include <stdio.h>
void main()
{
  printf("Oi");
}
```

→ Saída do programa:

```
Oi
```

Ex_2:

```
#include <stdio.h>
void main()
{
  printf("Primeiro ");
  printf("programa");
}
```

→ Saída do programa:

```
Primeiro programa
```

Note que o programa anterior (Ex_2) imprimiu numa mesma linha as mensagens que estavam em linhas de comando diferentes. Isto acontece porque não foram incluídos caracteres de controle, chamados de caracteres de “escape” dentro da mensagem `printf()`.

A tabela 2.1 lista algumas seqüencias de caracteres de escape que podem ser utilizadas com o função `printf()`.

Sequência de Escape	Resultado
\a	Caractere de Beep
\b	Caractere de Backspace
\n	**Caractere de Mudança de Linha**
\t	Caractere de Tabulação Horizontal
\'	Caractere de Aspas Simples
\"	Caractere de Aspas Duplas

Tabela 2.1: Sequências de escape para `printf()`.

Desta forma, para corrigir o problema verificado no Ex_2 deveríamos digitar:

Ex$_3$:

```
#include <stdio.h>
void main()
{
  printf("Primeiro\n");
  printf("programa");
}
```

→

Saída do programa:

```
Primeiro
programa
```

Outro exemplo – Ex$_4$:

```
#include <stdio.h>
void main()
{
  printf("Linha1\nLinha 2\nLinha
3\n");
}
```

→

Saída do programa:

```
Linha 1
Linha 2
Linha 3
```

A função `printf()` também pode imprimir dados numéricos, caracteres e strings. Esta função é dita de saída formatada pois os dados de saída podem ser formatados (alinhados, com número de dígitos variáveis, etc.) de acordo com os *caracteres_controle* utilizados,neste caso específico, os "especificadores de formato" – conforme mostra a tabela 2.2 à seguir.

Especificador de Formato	Tipo de Valor
%d	Valor inteiro
%i	Valor inteiro
%u	Valor inteiro sem sinal (*unsigned*)
%o	Valor inteiro na base octal
%x ou %X	Valor inteiro em Hexadecimal
%f	Valor em ponto flutuante (float)
%e ou %E	Ponto flutuante em notação exponencial
%c	Caracter no formato ASCII
%s	String de caracteres com terminador nulo
%%	Exibe o caracter de sinal percentual

Tabela 2.2: Especificadores de formato de saída para função `printf()`.

A figura 2.3 mostra a sintaxe correta para trabalhar com especificadores de formato.

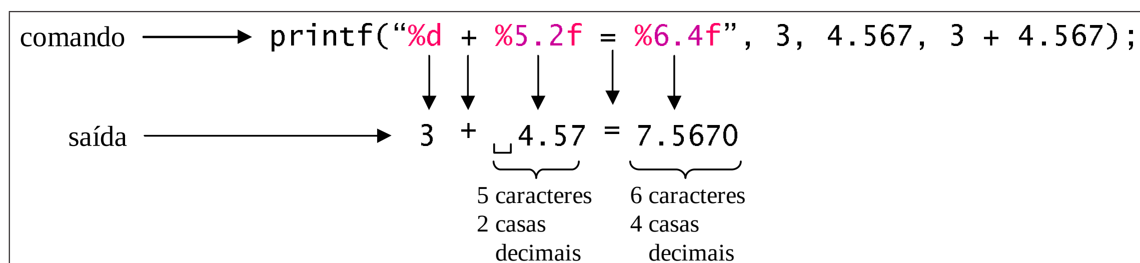


Figura 2.3: Exemplo de saída formatada com `printf()`.

Seguem programas exemplo mostrando o uso de especificadores de formato com a função `printf()`.

Programa 1:

prog1.c

```
#include <stdio.h>
void main()
{
   int a=3;
   int b=4;
   printf("a + b = %d + %i = %d", a, b, a+b);
}
```

Saída do programa anterior:

```
a + b = 3 + 4 = 7
```

Programa 2:

prog2.c
```
#include <stdio.h>
#include <conio.h> /* biblioteca que contem a função clrscr() */
void main() {
   int a=3;
   float b=4.5;
   clrscr(); /* esta funcao limpa a tela (clear screen) */
   printf("a + b = %d + %i = %d", a, b, a+b);
}
```

Saída do programa:

```
a + b = 3 + 0 = 0
```

Perceba que neste caso, foi mostrado um valor incorreto para a variável *b*. Isto ocorreu porque o programador usou `%d` para tentar exibir valores em ponto flutuante. Por conta disto a função `printf()` mostrou um resultado incorreto na tela. O correto seria codificar:

Programa 3:

prog3.c
```
#include <stdio.h>
#include <conio.h> /* contem a funcao clrscr() */
void main() {
   int a=3;
   float b=4.5;
   clrscr(); /* esta funcao limpa a tela (clear screen) */
   printf("a + b = %d + %f = %d", a, b, a+b);
}
```

Resultado da execução do programa:

```
a + b = 3 + 4.500000 = 0
```

Note que agora o conteúdo da variável *b* é exibido de forma correta mas o resultado da operação de adição não porque o especificador de de formato utilizado para demonstrar o resultado da adição continuou sendo `%d` quando deveria ter sido usado `%f` (o resultado da adição de um *int* + *float* retorna um *float*).

O programa a seguir explora um pouco mais as opções para formatação de dados de saída de um programa e as diferentes tipos de dados possíveis de serem declarados em ANSI C.

Programa 4:

prog4.c
```
/* Program 4 - LOTTYPES.C */
#include <stdio.h>
void main() {
  int a;              /* simple integer type                  */
  long int b;         /* long integer type                    */
  short int c;        /* short integer type                   */
  unsigned int d;     /* unsigned integer type                */
  char e;             /* character type                       */
  float f;            /* floating point type                  */
  double g;           /* double precision floating point      */

   a = 1023;
   b = 2222;
   c = 123;
   d = 1234;
   e = 'X';
   f = 3.14159;
```

```
    g = 3.1415926535898;

    printf("a = %d\n", a);      /* saída inteira                          */
    printf("a = %o\n", a);      /* saída em octal                         */
    printf("a = %x\n", a);      /* saída em hexadecimal                   */
    printf("b = %ld\n", b);     /* saída inteira formato longo            */
    printf("c = %d\n", c);      /* saída inteira formato curto            */
    printf("d = %u\n", d);      /* saída inteiro sem sinal                */
    printf("e = %c\n", e);      /* saída para caracter                    */
    printf("f = %f\n", f);      /* saída para ponto flutuante             */
    printf("g = %f\n", g);      /* saída para ponto flutuante dupla precisão */

    printf("\n");
    printf("a = %d\n", a);      /* saída para inteiro simples              */
    printf("a = %7d\n", a);     /* saída com largura de 7 caracteres       */
    printf("a = %-7d\n", a);    /* saída justificada à esquerda, 7 caracteres */

    c = 5;
    d = 8;
    printf("a = %5d\n", c, a);  /* use 5 caracteres de largura */
    printf("a = %8d\n", d, a);  /* use 8 caracteres de largura */

    printf("\n");
    printf("f = %f\n", f);      /* saída simples ponto flutuante    */
    printf("f = %12f\n", f);    /* saída com 12 caracteres          */
    printf("f = %12.3f\n", f);  /* saída com 3 casas decimais       */
    printf("f = %12.5f\n", f);  /* use 5 casas decimais             */
    printf("f = %-12.5f\n", f); /* saída alinhada à esquerda        */
    printf("f = %e\n", f);      /* saída formato científico         */
    printf("f = %5.2e\n", f);   /* saída formatada científica       */

}
```

Saída deste programa:

```
a = 1023
a = 1777
a = 3ff
b = 2222
c = 123
d = 1234
e = X
f = 3.141590
g = 3.141593

a = 1023
a =    1023
a = 1023
a =  1023
a =     1023

f = 3.141590
f =     3.141590
f =        3.142
f =      3.14159
f = 3.14159
f = 3.141590e+00
f = 3.14e+00
```

2.6.2 Comandos para Entrada de Dados

Função `scanf()`

A função `scanf()` é para entrada de dados à partir do teclado, implementada em todos os compiladores C (por padrão). Ela seria o complemento para a função `printf()`. Sua sintaxe é similar à `printf()`:

Uso: `scanf("`*expressão_de_controle*`", `*argumentos*`);`

onde "*expressão_de_controle*" serve para indicar o tipo de dado à ser lido, conforme mostra a tabela 2.3 à seguir:

Expressão de Controle	Tipo de dado à ser lido
`%d`	valor inteiro
`%f`	valor com casas decimais (ponto flutuante)
`%c`	apenas 1 caracter
`%s`	string (conjunto de caracteres)

Tabela 2.3: Expressões de controle para função `scanf()`.

e "*argumentos*" contem o nome da variável onde o dado lido através do teclado vai ser depositado. Em "argumentos" pode ser colocada uma lista de dados de entrada à serem entrados, no entanto, não se recomenda esta opção pois o sistema não "avisa" o usuário de que está esperando mais dados de entrada, e cada dado de entrada deve vir separado por vírgula ou espaço em branco. **Detalhe extra**: Antes do nome da varíavel em "*argumentos*", deve aparecer o símbolo `&` (que na realidade serve para avisar o compilador o endereço na memória onde deve guardar a informação lida do teclado).

Ex_1:

ent1.c
```
#include <stdio.h>
#include <conio.h>
void main() {
   int a, b;
   clrscr();
   printf ("Entre com a: ? ");
   scanf  ("%d", &a);
   printf ("Entre com b: ? ");
   scanf  ("%d", &b);
   printf ("\na + b = %d + %d = %d", a, b, a+b);
}
```

Saída do programa:

```
Entre com a: ? 3 Entre com b: ? 4

a + b = 3 + 4 = 7
```

2.6.3 Comentários dentro de um programa

Os pares de caracteres "`/*`" e "`*/`" delimitam blocos de comentários em C, sendo que esses comentários podem se estender por diversas linhas. A outra forma de comentário aceita pela linguagem é a barra dupla, "`//`", que inicia um comentário que termina com o final da linha.

O uso de comentários deve ser usado como forma de documentação interna de um programa. O excesso de comentários não é recomendado, pois pode até prejudicar a leitura de um programa. Entretanto, há comentários que devem estar presentes em qualquer programa, tais como:

Comentários de prólogo (ou iniciais): que aparecem no início de cada módulo ou arquivo-fonte. Devem indicar a finalidade do módulo, uma história de desenvolvimento (autor e data de criação e modificações), e uma descrição das variáveis globais (se houver);
Exemplo:

```
/* Programa: exemplo.c
   Programa para converter de graus Farenheit para graus Celcius
   Fernando Passold, em 16/03/2005
*/
```

Comentários de procedimento: que aparecem antes da definição de cada função indicando seu propósito, uma descrição dos argumentos e valores de retorno;

Comentários descritivos: que descrevem blocos de código. Comentários devem ser facilmente diferenciáveis de código, seja através do uso de linhas em branco, seja através de tabulações. É importante que comentários sejam corretos e coerentes com o código, uma vez que um comentário errôneo pode dificultar mais ainda o entendimento de um programa do que se não houvesse comentário nenhum.

2.6.4 EXERCÍCIOS

1) Escreva um programa para calcular a média de 3 valores informados pelo usuário.

2) Monte um programa capaz de calcular o valor final atingido por uma mercadoria, dado que o usuário informa o valor inicial da mercadoria e a porcentagem de desconto dada pela loja.

3) Escreva um programa capaz de calcular as raízes para uma equação do 2º grau: $ax^2 + bx + c = 0$, onde o usuário entra com os coeficientes da equação: a, b e c.

4) Monte um programa que converta uma temperatura em graus Farenheit fornecida pelo usuário para graus Célcius, sabendo-se que:

$$^oC = \frac{5\,(^oF - 32)}{9}$$

2.6.5 PROBLEMAS RESOLVIDOS

Segue uma lista de pequenos programas realizáveis em sala de aula.

Primeiro exemplo:

TESTE1.CPP

```
/* TESTE1.CPP
   Simples programa para mostrar função printf()
*/
#include <stdio.h>
void main(){
  clrscr();
  printf("Mensagem");
  printf("Mensagem2");
}
```

Saída do programa:

```
MensagemMensagem2
```

Note que apesar de terem sido usadas duas funções `printf()`, as mensagens não apareceram em linhas separadas e sim, uma após a outra. Para fazer com que as mensagens aparecessem cada uma na sua linha, deveríamos ter usado o caracter de escape "\n" na primeira função `printf()`, como demonstra o programa à seguir: TESTE2.CPP.

TESTE2.CPP

```
/* TESTE2.CPP
   Usando caracteres de escape com a função printf()
*/
#include <stdio.h>
#include <conio.h> /* biblioteca com função clrscr() */
void main()
{
  clrscr(); /* esta função limpa a tela (clear screen) */
  printf("Mensagem\n");
```

```
  printf("Mensagem2");
}
```

Saída do programa:

```
Mensagem Mensagem2
```

Note que agora, cada mensagem apareceu na sua linha.

Mensagens em linhas separadas não necessariamente devem usar vários comandos `printf()` diferentes:

TESTE4.CPP

```
/* TESTE4.CPP
   Brincando um pouco mais com \n e \t na função printf()
*\
#include <stdio.h>
#include <conio.h>
void main(){
  clrscr();
  printf("\n\n\nMensagem\tMensagem2");
}
```

Que gera o resultado:

```



Mensagem        Mensagem2
```

Note que pulamos 2 linhas na tela antes da primeira mensagem.

Outro exemplo, mas agora usando também especificadores de formato para mostrar informações numéricas na tela:

TESTE6.CPP

```
/* TESTE6.CPP */
#include <stdio.h>
#include <conio.h>
void main(){
  clrscr();
  printf("\na + b = %d + %d = %d", 3, 4, 3+4);
}
```

gera a saída:

```

 a + b = 3 + 4 = 7
```

Especificando agora diferentes formatos de saída de acordo com o tipo de certas variáveis:

TESTE7.CPP

```
/* TESTE7.CPP */
#include <stdio.h>
#include <conio.h>
void main(){
  int a;
  float b;
  int c;
  a=3;
  b=4.57656566855;
  c=a+b;
  clrscr();
  printf("a + b = \%d + \%d = \%d", a, b, c);
}
```

Note que este programa gera a saída:

```
a + b = 3 + 0 = 16384
```

quando deveria gerar algo como:

```
a + b = 3 + 4.57656566855 = 7.57656566855
```

O programa anterior possui alguns **erros de programação**:

1) O primeiro erro (do programador) foi não ter percebido que a variável 'b' na linha 06 foi declarada como *float*. Lembrando que: *int* + *float* → *float*, a variável 'c' na linha 07 também deveria ter sido declarada como *float*.

2) E na linha 12, os especificadores de formato usados para imprimir o conteúdo das variáveis 'b' e 'c' estão errados. Deveria ter sido usado %f (adequado para variáveis do tipo *float*) no lugar de %d (apropriado para variáveis do tipo *int*).

3) **Erro de execução**: Tentar imprimir o conteúdo de uma variável do tipo *float* usando especificador para variável do tipo *int*, faz com que o compilador gere informação incorreta na tela: 0 para o conteúdo da variável 'b' e 16384 para o conteúdo da variável 'c'.

O programa anterior corrigido fica:

TESTE8.CPP

```
/* TESTE8.CPP - correção do TESTE7.CPP */
#include <stdio.h>
#include <conio.h>
void main(){
  int a;
  float b, c;
  a=3;
  b=4.57656566855;
  c=a+b;
  clrscr();
  printf(" a + b = %d + %f = %f", a, b, c);
}
```

Que, agora sim, gera a saída:

```
a + b = 3 + 4.576566 = 7.576566
```

Note porém que as variáveis do tipo flutuante foram impressas com várias casas após a vírgula. Isto pode ser "consertado" se nos espeficadores de formato adotados para as variáveis do tipo *float* houvesse sido indicado a quantidade total de caracteres e número de casas decimais desejadas, como demonstra o programa à seguir:

TESTE9.CPP

```
/* TESTE9.CPP - melhoria do TESTE8.CPP */ #include <stdio.h>
#include <conio.h> void main(){
  int a;
  float b, c;
  a=3;
  b=4.57656566855;
  c=a+b;
  clrscr();
  printf(" a + b = %d + %4.2f = %4.2f", a, b, c);
}
```

Que gera a saída:

```
a + b = 3 + 4.58 = 7.58
```

Mas o que aconteceria se o conteúdo da variável 'b' do programa anterior fosse negativo?

TESTE10.CPP

```
1 /* TESTE10.CPP - variação do TESTE9.CPP */
2 #include <stdio.h>
3 #include <conio.h>
4 void main(){
5   int a;
6   float b, c;
7   a=3;
8   b= -4.57656566855; /* Unica diferenca para TESTE9.CPP */
9   c=a+b;
10  clrscr();
11  printf(" a + b = \%d + \%4.2f = \%4.2f", a, b, c);
12 }
```

Seria gerada a saída:

```
a + b = 3 + -4.58 = -1.58
```

Note, que não havia sido previsto no TESTE9.CPP, o caso de alguma das varáveis ser negativo. Note ainda que nos espcificadores de formato para o conteúdo destas variáveis, temos que prever uma quantidade extra de caracteres, para o caso de ser necessário imprimir o próprio sinal negativo do número. Sendo assim, a linha 11 deveria ser corigida para:

11. `printf("a + b = %d + %5.2f = %5.2f", a, b, c);`

Parte II

Estruturas de Controle de Fluxo

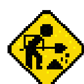
Curso de Engenharia Elétrica
Informática Aplicada à Engenharia Elétrica I

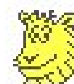
2ª Parte da Apostila de ANSI C

Estruturas de Controle de Fluxo

Prof. Fernando Passold

Observação: Material em fase de edição
Usando sistema de edição MiKTeX/LaTeX 2ε para a mesma:
Ver: http://www.miktex.org/
Prof. Fernando Passold – Semestre 2005/1
Última atualização: 13 de março de 2006.





Capítulo 3

Estruturas de Decisão

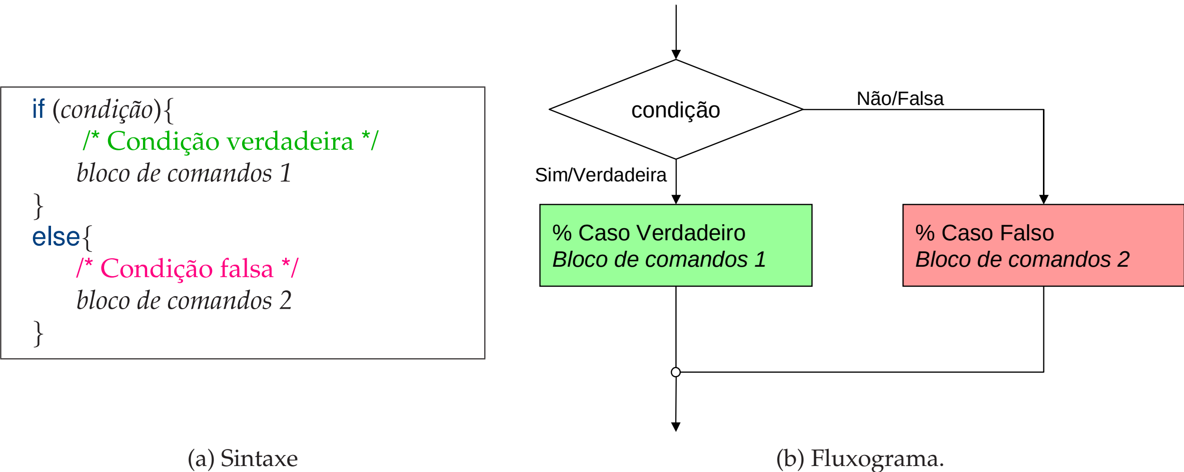
Contents

Neste capítulo você aprenderá a usar as estruturas de decisão do C para alterar o fluxo de execução de um programa conforme o resultado de condições sendo testadas. Serão estudados os comandos:

- if (*condição*) { *bloco de comandos 1* } else { *bloco de comandos 2* }
- switch (*variável*){ case (*constante1*): ..; break; ... default: ...;}

3.1 Comando if..else

Note que para resolver certos algoritmos faz falta realizar um teste na condição de certas variáveis e tomar uma decisão entre mais de uma opção de acordo com o resultado deste teste. A figura 3.1 mostra a sintaxe a na forma de fluxograma o comando if..else.. da linguagem C.



(a) Sintaxe (b) Fluxograma.

Figura 3.1: Comando if..else...

A *condição* à ser testada pode ser qualquer expressão usando operadores chamados "relacionais" (porque permitem testar uma relação). A tabela 3.1 mostra uma lista destes operadores disponíveis na linguagem C.

Operador Matemático	Função	Operador na Linguagem C	Exemplo em C
$>$	Maior que	`>`	`(a > b)`
$<$	Menor que	`<`	`((a+b)<c)`
$\geq$	Maior ou igual a	`>=`	`a >= b`
$\leq$	Menor ou igual a	`<=`	`a <= 0`
$=$	Igual	`==`	`a == b`
$\neq$	Diferente	`!=`	`a != 0`

Tabela 3.1: Tabela de Operadores Relacionais

Este comando funciona assim, caso a *condição* testada seja verdadeira (ou um valor $\neq 0$), o *bloco de comandos 1* é executado, senão (else) (ou valor do teste = 0), o *bloco de comandos 2* é executado.

Observações:

1. Se o teste à ser executado é simples (1 única linha de comandos), as chaves ("{" e "}") não são necessárias.
2. O else do comando if é opcional.

Em PORTUGOL este comando ficaria:

```
SE (condição) ENTÃO
    INICIO
        bloco de comandos 1
    FIM
SENÃO
    INICIO
        bloco de comandos 2
    FIM
```

Exemplos

Exemplo$_1$

```c
void main(){
  int value = 5;

  if (value >= 0 )
    printf ("Valor eh positivo\n");

  printf("Valor eh %d", value);
}
```

Quando é executado o comando:

```c
  if (value >= 0 )
```

o conteúdo da variável `value` é testada. Se `value` é maior ou igual à zero, o comando seguinte é executado, ou seja, a linha:

```c
    printf ("Valor eh positivo\n");
```

Se `value` é menor que 0, o programa mostraria apenas o segundo comando *printf*, ou seja, executaria a linha:

```c
  printf("Valor eh %d", value);
```

Note ainda que um ponto-e-vírgula não finaliza a linha contendo o comando if:

```
4   if (value >= 0 )
```

Um comando *if* realmente contém sempre no mínimo duas partes. A primeira especifica a condição e a segunda contém o comando que será executado se a condição for verdadeira. Como pode ser notado, o bloco de comandos referente ao caso do teste ser verdadeiro, a parte do *printf* identado, linha 5, não inicia nem termina com chaves, porque neste caso de uso do *if*, existe apenas 1 comando para ser executado – lembre-se da observação 1.

Também repare que o comando associado a *if* está identado. Embora o compilador C não obrigue a identação, os programadores costumam identar comandos associados com uma instrução para auxiliar visualmente o programador.

O exemplo anterior pode ser ampliado, encaixando os símbolos { e } para indicar o início e o fim do bloco de comandos referente ao caso da condição testada ser verdadeira:

```
1 void main(){
2   int value = 5;
3
4   if (value >= 0 ){
5     printf ("Valor eh positivo\n");
6   }
7   printf("Valor eh %d", value);
8 }
```

Exemplo$_2$

```
#include <stdio.h>
void main(){
   int value;
   printf("\nEntre com um valor inteiro: ? ");
   scanf("%i", &value);
   if (value >= 0)
      {
      printf("O valor digitado eh positivo\n");
      }
   else
      {
      printf("O valor digitado eh negativo\n");
      }
   printf("Fim.");
}
```

Este programa pode gera a seguinte saída:

```
Entre com um valor inteiro: ? 3
O valor digitado eh positivo
Fim.
```

caso o usuário tenha entrado com um valor ≥ 0. Ou gerar uma saída como a mostrada abaixo em caso contrário:

```
Entre com um valor inteiro: ? -5
O valor digitado eh negativo
Fim.
```

O programa anterior poderia ser escrito de uma maneira mais resumida:

```c
#include <stdio.h>
void main(){
   int value;
   printf("\nEntre com um valor inteiro: ? ");
   scanf("%i", &value);
   if (value >= 0)
      printf("O valor digitado eh positivo\n");
   else
      printf("O valor digitado eh negativo\n");
   printf("Fim.");
}
```

e os mesmos resultados seriam gerados (comprove na prática).

O Fluxograma equivalente ao programa acima aparece na figura 3.2.

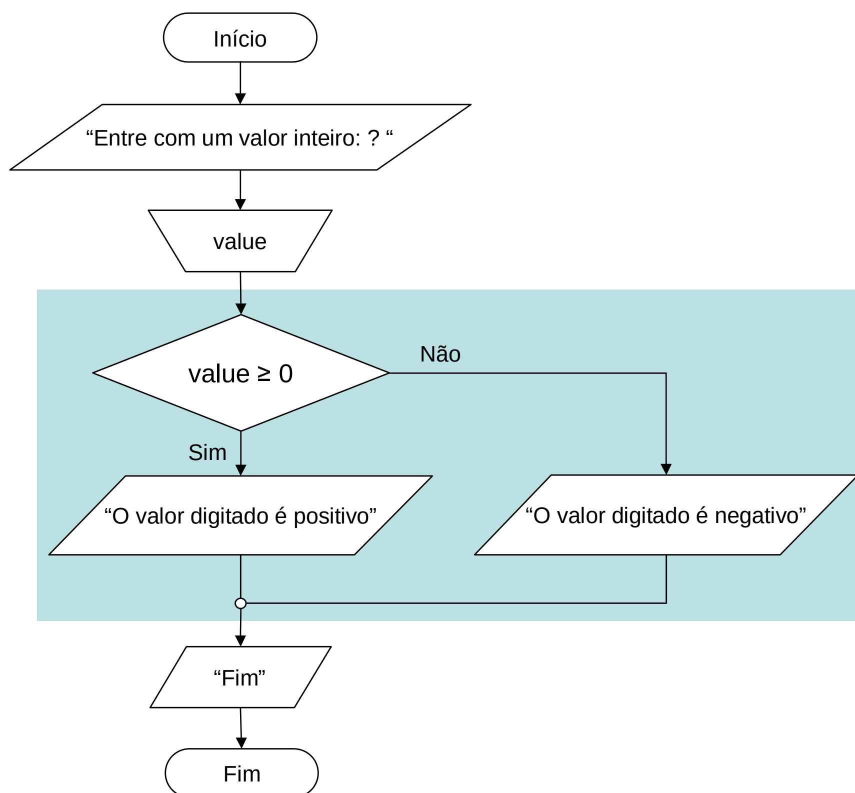


Figura 3.2: Fluxograma do Exemplo$_2$.

Problema:

E o que aconteceria se o usuário digitasse 0 (zero)??? Como este problema poderia ser resolvido?

Detalhes:



A linguagem C SEMPRE avalia qualquer valor diferente de zero como verdadeiro.
Por exemplo:
if (0) ⇐ Sempre resultará como um teste FALSO;
if (53) ⇐ Sempre resultará como um teste VERDADEIRO;

Por exemplo:

```
#include <stdio.h>
#include <conio.h>
void main(){
  int i;
  clrscr();

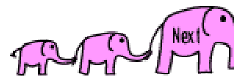
  if (0)
    printf("Verdadeiro\n");
  else
    printf("Falso\n");

  if (53)
    printf("Verdadeiro\n");
  else
    printf("Falso\n");

}
```

Saída:

```
Falso
Verdadeiro
```



Exemplo$_3$

Por exemplo, um simples programa escrito em ANSI C para calcular as raízes de uma equação do 2° grau só funciona para os casos em que suas raízes sejam reais ($x_i \in \Re$), mas dá erro no caso de raízes complexas, por exemplo:

```
Programa Calcula raízes equação 2o grau:

Formato: a*x^2 + b*x + c = 0

Entre com a: ? 2 Entre com b: ? 2 Entre com c: ? 2

Delta = -12.000000 sqrt: DOMAIN error sqrt: DOMAIN error

x1 = +NAN x2 = +NAN
```

Note que o programa consegue determinar o valor de $\triangle = b^2 - 4ac$, mas a função `sqrt()` não está preparada para extrair a raiz quadrada de números negativos, o que resula nas mensagens de erro:

`sqrt: DOMAIN error sqrt: DOMAIN error`

e

`x1 = +NAN x2 = +NAN`

referentes às linhas 23 à 25 do programa abaixo:

```
1 /* Programa equacao do 2o grau - 1a versão
2    Fernando Passold, em 23/03/2005
3    */
4 #include <stdio.h>
5 #include <conio.h> /* contem clrscr() */
6 #include <math.h>  /* contem sqrt()   */
```

```
void main(){
   /* Declaracão de variáveis */
   float a,b,c,delta,x1,x2;
   clrscr();
   /* Etapa de entrada de dados */
   printf("Programa Calcula raízes equação 2o grau:\n\n");
   printf("Formato: a*x^2 + b*x + c = 0\n\n");
   printf("Entre com a: ? ");
   scanf("%f", &a);
   printf("Entre com b: ? ");
   scanf("%f", &b);
   printf("Entre com c: ? ");
   scanf("%f", &c);
   /* Etapa de processamento */
   delta=b*b-4*a*c;
   printf("\nDelta = %f\n", delta);
   x1=(-b+sqrt(delta))/(2*a);
   x2=(-b-sqrt(delta))/(2*a);
   printf("\nx1 = %f\nx2 = %f\n", x1,x2);
   getch(); /* "congela" tela ate usuário apertar uma tecla */
}
```

Seria muito interessante se houvesse uma forma do programa avisar o usuário de que as raízes do seu sistema são complexas e parar o cálculo das raízes ou tentar realizar este cálculo com alguma modificação no programa mostrado anteriormente.

Isto pode ser feito se houver uma forma de testar o valor de $\triangle$ e então direcionar o programa para continuar realizando o cálculo ou não. Para tanto, necessitamos de um comando que possibilite esta "tomada de decisão". Este comando é o "IF..THEN.." (ou "SE..ENTÃO.." em PORTUGOL), ou "if" em linguagem C.

Solução$_1$: em PORTUGOL:

```
INICIO
    LER a,b,c
    delta = b*b-4*a*c
    IF  delta < 0 ENTÃO
    INÍCIO
      IMPRIMIR "Não há raízes reais."
    FIM
    SENÃO
    INÍCIO
      x1 = (-b + sqrt(delta))/(2*a)
      x2 = (-b + sqrt(delta))/(2*a)
      IMPRIMIR x1, x2
    FIM
FIM
```

Solução em C:

```
/* Programa equação do 2o grau - 2a versão
   Fernando Passold, em 23/03/2005
*/ #include <stdio.h> #include <conio.h> /* contem clrscr() */
#include <math.h>  /* contem sqrt()   */ void main(){
   /* Declaracão de variáveis */
   float a,b,c,delta,x1,x2;
   clrscr();
   /* Etapa de entrada de dados */
   printf("Programa Calcula raízes equação 2o grau:\n\n");
   printf("Formato: a*x^2 + b*x + c = 0\n\n");
   printf("Entre com a: ? ");
   scanf("%f", &a);
   printf("Entre com b: ? ");
   scanf("%f", &b);
   printf("Entre com c: ? ");
   scanf("%f", &c);

   /* Etapa de processamento */
   delta=b*b-4*a*c;
   printf("\nDelta = %f\n", delta);

   /* Etapa de Saída de Dados */
   if (delta < 0)
      printf("Não há raízes reais\n");
   else {
      x1=(-b+sqrt(delta))/(2*a);
      x2=(-b-sqrt(delta))/(2*a);
      printf("\nx1 = %f\n", x1);
      printf("x2 = %f\n",   x2);
   }
   getch(); /* "congela" tela ate usuário pressionar uma tecla */
}
```

Saída do programa:

```
Programa Calcula raízes equação 2o grau:

Formato: a*x^2 + b*x + c = 0

Entre com a: ? 2 Entre com b: ? 2 Entre com c: ? 2

Delta = -12.000000 Não há raízes reais
```

Solução$_2$:

O tipo de solução anterior pode não satisfazer o usuário. Entretanto o ANSI C não permite trabalhar com variáveis complexas. A solução então é "improvisar". Note que no caso do delta ser negativo, as raízes se decompõem na sua parte Real e na sua parte Imaginária, da seguinte forma:

$$x_1 = \frac{-b}{2a} - j\frac{\sqrt{-\Delta}}{2a}$$

e

$$x_2 = \underbrace{\frac{-b}{2a}}_{\text{Real}} + \underbrace{j\frac{\sqrt{-\Delta}}{2a}}_{\text{Imaginaria}}$$

Note que já que a função `sqrt()` do ANSI C não extrai a raiz quadrada de números negativos (resulta num número complexo), multiplicamos o Δ encontrado anteriormente por -1, tornado assim este número positivo. Só temos que nos lembrar que: $\sqrt{-1} = -j$. Dai os 2 termos encontrados para a raíz x_1 e x_2. Como cada uma das raízes possui então dois componentes, a parte real e a parte imaginária, declaramos 2 variáveis para cada uma das raízes, por exemplo: `x1_Real`, `x1_Imag`, `x2_Real`, e `x2_Imag` e o problema estaria resolvido. Veja o código abaixo:

```
/* Programa equação do 2o grau - 3a versão
    Fernando Passold, em 23/03/2005
*/
#include <stdio.h>
#include <conio.h> /* contem clrscr() */
#include <math.h>  /* contem sqrt()   */
void main(){
    /* Declaracão de variáveis */
    float a,b,c,delta,x1,x2;
    float x1_Real,x1_Imag,x2_Real,x2_Imag;
    clrscr();
    /* Etapa de entrada de dados */
    printf("Programa Calcula raízes equação 2o grau:\n\n");
    printf("Formato: a*x^2 + b*x + c = 0\n\n");
    printf("Entre com a: ? ");
    scanf("%f", &a);
    printf("Entre com b: ? ");
    scanf("%f", &b);
    printf("Entre com c: ? ");
    scanf("%f", &c);

    /* Etapa de processamento */
    delta=b*b-4*a*c;
    printf("\nDelta = %f\n", delta);

    /* Etapa de Saída de Dados */
    if (delta < 0){
        x1_Real=-b/(2*a);
        x1_Imag=sqrt(-delta)/(2*a);
        x2_Real=-b/(2*a);
        x2_Imag= -sqrt(-delta)/(2*a);
        printf("\nx1 = %f + j%f\n", x1_Real,x1_Imag);
        printf("x2 = %f -j%f\n", x2_Real,x2_Imag);
    }
    else {
        x1=(-b+sqrt(delta))/(2*a);
        x2=(-b-sqrt(delta))/(2*a);
        printf("\nx1 = %f\n", x1);
        printf("x2 = %f\n",   x2);
    }
    getch(); /* "congela" tela ate usuário apertar uma tecla */
}
```

Saída do programa:

```
Programa Calcula raízes equação 2o grau:

Formato: a*x^2 + b*x + c = 0

Entre com a: ? 2 Entre com b: ? 2 Entre com c: ? 2

Delta = -12.000000

x1 = -0.500000 + j0.866025 x2 = -0.500000 -j-0.866025
```

O único problema continua sendo a forma de apresentar os resultados na tela, especialmente para o caso da parte complexa ser negativa. Você saberia como melhorar a saída anterior?

Problemas

Problema$_1$: Monte um programa onde o usuário entra com 2 números quaisquer e o programa indica depois qual foi o maior valor entrado.

Problema$_2$: Melhore a saída da 3a versão do programa para calcular as raízes de uma equação do 2o grau, no caso de raízes complexas, de forma a evitar que o mesmo gere uma saída como:

```
x1 = -0.500000 + j0.866025 x2 = -0.500000 -j-0.866025
```

3.2 Usando operadores Lógicos

Existem 3 operadores lógicos em C que permitem ampliar a condição testada: AND (`&&`), OR (`||`) e NOT (`!`), conforme demonstram a tabela 3.2 à seguir.

Operador em Linguagem Natural	Operador na Linguagem C	Função	Exemplo
E	`&&`	Lógico AND	`if (letter >= 'a' && letter <= 'z')`
Ou	`||`	Lógico OR	`if (letter == 'y' || letter == 'Y')`
Negação	`!`	Lógico NOT	`if (!((letter == 'y') || (letter == 'Y')))`

Tabela 3.2: Operadores lógicos em C.

Exemplos

Exemplo$_1$: O programa abaixo indica se um aluno está em EXAME depois que o mesmo entrou com sua média semestral.

EXAME.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
   float media;

   clrscr();
   printf("Entre com a media semestral: ? ");
   scanf("%f", &media);

   printf("Este aluno ");
   if ((media>=3)&&(media<7))
      printf("ESTA' EM EXAME\n");
   else
```

```
      printf("nao esta' em exame\n");

   getch();
}
```

Rode este programa e verifique as saídas geradas.

3.3 Comandos if encadeados

Comandos if podem ser encadeados um dentro do outro. Por exemplo:

```
if y == 1
   Tarefa_a();
else
   if y == 2
      Tarefa_b();
   else
      if y == 0
         Tarefa_c();
      else
         Tarefa_d();
```

O segmento anterior de programa poderia também ter sido escrito na forma:

```
if y == 1
   Tarefa_a();
   else if y == 2
            Tarefa_b();
            else if y == 0
                     Tarefa_c();
                     else
                     Tarefa_d();
```

Na forma de fluxograma, o segmento de programa acima ficaria como o mostrado na figura ao lado.

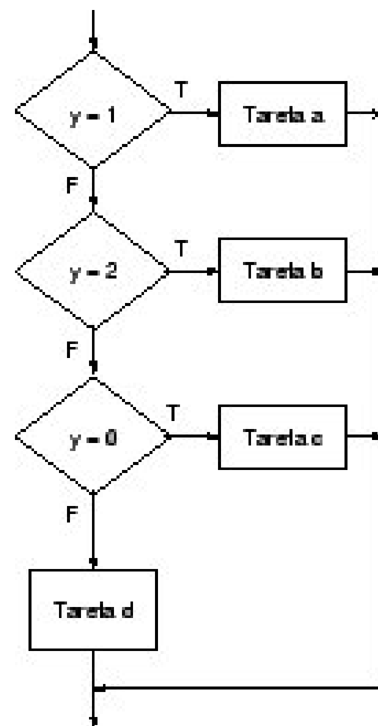


Figura: Exemplo de IF's encadeados.

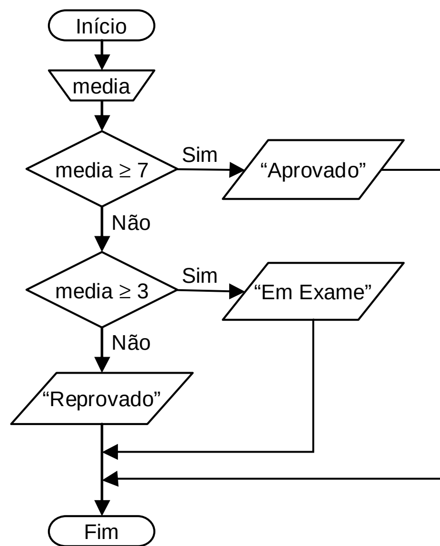
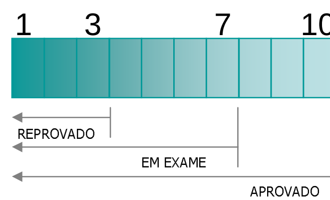


Figura 3.3: Fluxograma relativo à solução 1 do Ex.$_1$.

Exemplo

Monte um programa para indicar se um aluno foi “Aprovado”, está em “Exame” ou foi “Reprovado” numa disciplina, baseado na média semestral informada pelo usuário.

Solução$_1$: Comparando a nota obtida pelo aluno do valor mais baixo para o mais alto:



O fluxograma representando esta solução é mostrado na figura 3.3.

O código referente ao fluxograma da figura 3.3 é mostrado abaixo:

MEDIA1.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
   float media;

   clrscr();
   printf("Entre com a media semestral: ? ");
   scanf("%f", &media);

   if (media >= 7)
      printf("O aluno foi APROVADO\n");
   else{
      if (media >= 3)
         printf("O aluno está em EXAME\n");
      else
         printf("O aluno foi REPROVADO\n");
   }
   getch();
}
```

Solução$_2$: Outra solução:

MEDIA2.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
   float media;

   clrscr();
   printf("Entre com a media semestral: ? ");
   scanf("%f", &media);

   if (media >= 7)
      printf("O aluno foi APROVADO\n");
   else{
      if (media < 3)
         printf("O aluno foi REPROVADO\n");
      else
         printf("O aluno esta’ em EXAME\n");
   }
   getch();
}
```

Solução$_3$: E se o problema anterior tivesse sido resolvido da forma abaixo. Onde está o erro?

MEDIA3.CPP

```
1  #include <stdio.h>
2  #include <conio.h>
3  void main(){
4     float media;
5
6     clrscr();
7     printf("Entre com a media semestral: ? ");
8     scanf("%f", &media);
9
10    printf("Este aluno está");
11    if (media >= 3)
12       printf("em EXAME\n");
13    else
14       if (media >= 7)
15          printf("APROVADO\n");
16       else
17          printf("REPROVADO\n");
18
19    getch();
20 }
```

Exemplo de saída gerada:

```
Entre com a media semestral: ? 8
Este aluno está em EXAME
```

Note que o erro está na lógica de programação envolvendo os IF’s das linhas 11 e 14. Note que o primeiro IF, da linha 11, separa *media* em valores maiores ou iguais à 3 – o primeiro erro (conforme demonstra o exemplo de saída mostrado acima), caso contrário, o programa segue para o segundo IF (encadeado), da linha 14, mas quando alcança este IF, só sobraram os casos de *media* $<$ 3 e assim a condição `(media >= 7)` nunca vai ocorrer – o segundo erro de lógica de programação.

Solução$_4$: E se este problema tivesse sido resolvido desta forma:

MEDIA4.CPP
```
#include <stdio.h>
#include <conio.h>
void main(){
   float media;

   clrscr();
   printf("Entre com a media semestral: ? ");
   scanf("%f", &media);

   printf("Este aluno está ");
   if ((media >= 3)&&(media<7))
      printf("em EXAME\n");
   else
      if (media >= 7)
         printf("APROVADO\n");
      else
         printf("REPROVADO\n");

   getch();
}
```

Resposta: a lógica deste programa está correta apesar de terem sido utilizados operadores lógicos.

Solução$_5$: E o que está errado na versão abaixo?

MEDIA5.CPP
```
1  #include <stdio.h>
2  #include <conio.h>
3  void main(){
4     float media;
5
6     clrscr();
7     printf("Entre com a media semestral: ? ");
8     scanf("%f", &media);
9
10    printf("Este aluno esta' ");
11    if (media>=7)
12       printf("APROVADO");
13    if (media<3)
14       printf("REPROVADO");
15    else
16       printf("em EXAME");
17
18    getch();
19 }
```

que é capaz de gerar uma saída como:

```
Entre com a media semestral: ? 8 Este aluno esta' APROVADOem EXAME
```

Desta vez, repare que os 2 IF's não estão encadeados entre si, isto é, o IF da linha 13 sempre será executado tal qual o IF da linha 11. Isto é que gera o erro como o mostrado no exemplo de saída demonstrado para este programa (acima).

Problemas

Problema$_1$: Monte um programa que classifique a faixa etária das pessoas conforme a idade que foi informada para elas, segundo a tabela abaixo:

Bebê	→	menos que 2 anos de vida
Criança	→	maior que 2 e até 12 anos
Adolescente	→	maior que 12 mas menor que 23
Adulto	→	maior que 23 mas menor que 70
Idoso	→	maior que 70

Possível Solução:

IDADES.CPP

```
/* Programa: idades.cpp */
/* Problema 1: classificacao de faixas etarias */
#include <stdio.h>
#include <conio.h>
void main(){
  float idade;
  clrscr();
  printf("Entre com a idade da pessoa: ? ");
  scanf("%f", &idade);

  if (idade <= 2)
     printf("é um bebê\n");
  else
     // if ((idade > 2)&&(idade <= 12)) <- AND redundante!
     if (idade <= 12)
        printf("é uma criança\n");
     else
        if (idade <= 23)
           printf("é um adolescente\n");
        else
           if (idade <= 70)
              printf("é um adulto\n");
           else
              printf("é um idoso\n");

  getch();
}
```

Note que o programa anterior pode também ser codificado da seguinte forma (usando os marcadores de início e fim para cada if e else: "{" e "}"):

IDADES.CPP

```
/* Programa: idades.cpp */
/* Problema 1: classificacao de faixas etarias */
#include <stdio.h>
#include <conio.h>
void main(){
  float idade;
  clrscr();
  printf("Entre com a idade da pessoa: ? ");
  scanf("%f", &idade);

  if (idade <= 2) {
     printf("é um bebê\n"); }
  else {
     // if ((idade > 2)&&(idade <= 12)) <- AND redundante!
     if (idade <= 12) {
        printf("é uma criança\n"); }
     else {
        if (idade <= 23) {
           printf("é um adolescente\n"); }
        else {
           if (idade <= 70) {
              printf("é um adulto\n"); }
```

```
            else {
               printf("é um idoso\n");
            }
         }
      }
   }
   getch();
}
```

Problema$_2$: Monte um programa capaz de identificar o tipo de triângulo baseado nas dimensões dos 3 lados passados pelo usuário. Sendo que o caso de um triângulo possuir 3 lados iguais, o qualifica como triângulo "equilátero"; no caso de possuir 2 lados iguais, é um triângulo "isósceles", senão é o caso de um triângulo "qualquer". E no caso do usuário informar algum dos lados maior que a soma dos outros 2 lados configura uma figura geométrica que não caracteriza um triângulo.

Possível Solução:

TRIANG.CPP

```
/* Programa: triang.c
   Problema 2: classificação de triângulos */
#include <stdio.h>
#include <conio.h>
void main(){
  float a, b, c;
  clrscr();
  printf("Entre com as dimensões dos lados de um triângulo:\n\n");
  printf("Lado a: ? ");
  scanf("%f", &a);
  printf("Lado b: ? ");
  scanf("%f", &b);
  printf("Lado c: ? ");
  scanf("%f", &c);

  if ( (a < (b+c)) && (b < (a+c)) && (c < (a+b)) ) {
     // é um triângulo de fato...
     if ((a == b) && (b == c)) {
        printf("O triângulo é equilátero");
     }
     else {
        if ( (a == b) || (a == c) || (c == b) ) {
           printf("O triângulo é isosceles");
        }
        else {
           printf("É um triângulo qualquer");
        }
     }
  }
  else {
     printf("Não é um triângulo");
  }

  getch();
}
```

Problema$_3$: Monte um programa onde o usuário entra agora com 3 números e que depois mostre o maior número entrado. Teste as 3 possibilidades possíveis, por exemplo:

a	b	c
5	4	3

a	b	c
4	5	3

a	b	c
4	3	5

Solução$_1$:

IFPROB3.CPP

```
/* Programa: IFPROB3.CPP
   Problema 3: classificação de números */
#include <stdio.h>
#include <conio.h>
void main(){
  float a, b, c;
  clrscr();
  printf("Entre com 3 numeros:\n\n");
  printf("a: ? ");
  scanf("%f", &a);
  printf("b: ? ");
  scanf("%f", &b);
  printf("c: ? ");
  scanf("%f", &c);

  if ( (a>=b) && (a>=c) ) {
     printf("Maior numero => a=%f\n", a);
  }
  else {   /* a < b e a < c */
     if ( b>=c ) {
    printf("Maior numero => b=%f\n", b);
     }
     else {
    printf("Maior numero => c=%f\n", c);
     }
  }
  getch();
}
```

O código acima é suficiente para descobrir qual o maior número, mas é incapaz de identificar o caso dos 3 números serem iguais ou os 2 maiores serem iguais. Para prever estes casos é necessário se refinar esta primeira solução.

Solução$_2$:

IFPROB3B.CPP

```
/* Programa: IFPROB3B.CPP - versão melhorada
   Problema 3: classificação de 3 números     */
#include <stdio.h>
#include <conio.h>
void main(){
  float a, b, c;
  clrscr();
  printf("Entre com 3 números:\n\n");
  printf("a: ? ");
  scanf("%f", &a);
  printf("b: ? ");
  scanf("%f", &b);
  printf("c: ? ");
  scanf("%f", &c);

  if ( (a==b) && (b==c) ){ /* Não necessito testar se a==c ! */
     printf("Os 3 números informados sao iguais\n");
  }
  else {
     if ( (a>=b) && (a>=c) ){
        /* mas a=b? ou a=c? */
        if (a==b){
           printf("Maiores números => a e b que sao iguais (%f)\n", a);
        }
        else{
           if (a==c){
              printf("Maiores números => a e c que sao iguais (%f)\n",a);
           }
           else{
              printf("Maior numero => a = %f\n", a);
           }
        }
     }
     else { /* a<b, a<c */
        if (b==c){
           printf("Maiores números => b e c, que sao iguais (%f)\n", b);
        }
        else {
           if (b>c){
              printf("Maior numero => b = %f\n", b);
           }
           else {
              printf("Maior numero => c = %f\n", c);
           }
        }
     }
  }

  getch();
}
```

Note que agora esta solução é capaz de prever casos como:

```
Entre com 3 números:

a ? 5
b ? 2
c ? 5

Maiores números => a e c que sao iguais (5.0000)
```

Observações:



O código da solução$_2$ demonstra ainda **como é importante começar a identar (tabular) corretamente o programa** até para melhor compreensão do mesmo ou no mínimo para o programador ter a certeza de onde iniciam e começam cada bloco de if { .. } e else { .. } .

Uma boa prática para evitar esquecimentos em relação à }'s principalmente (resulta em mensagens de erro do compilador como: "`Missplaced else`" – ou else "perdido"), é, durante a codificação do programa, o programador já ir se habituando a abrir ({) e fechar (}) cada bloco no momento em que está digitando cada if ou cada else.

Por exemplo:

```
if ( condição_testada ) {
    ...
}
else {
    ...
}
```



Problema$_4$: Complemente o problema anterior, mas agora, depois do usuário entrar com os 3 números, o programa deve mostrar os 3 números em ordem decrescente. Quantas possibilidades existem? Existe uma forma de montar um algoritmo bastante curto para executar esta tarefa?

3.4 Operador Ternário – Operador ?

C permite realizar testes IF simples usando o operador dito ternário: "?". É usado no lugar de declarações IF do tipo:

```
if (condição) {
    variável = expressão_1; }
else {
    variável = expressão_2;
}
```

Sintaxe: *variável* = *condição* ? *expressão_1* : *expressão_2*;

Se a *condição* testada for verdadeira, o valor da *expressão_1* é atribuído à *variável*; caso contrário, *variável* assume o valor de *expressão_2*.

Exemplo:

```
x = (y < 10) ? 20 : 40;
```

atribui a x, o valor 20 se y for menor que 10 ou atribui a x, o valor 40 caso y seja maior ou igual à 10.

3.5 Comando switch case (Decisões Múltiplas)

Adicionalmente ao comando IF..ELSE, programas em C podem usar o comando switch case para processamento de condições.

Sintaxe:

```
switch (variável) {
    case (constante1): { bloco de comandos 1 } break;
    case (constante2): { bloco de comandos 2 } break;
    ⋮
    default: { bloco de comandos → nenhum dos casos anteriores }
}
```

Este comando avalia o estado de uma variável e então tenta localizar uma coincidência dentro de uma lista de possibilidades digitadas pelo programador. Se a lista contém uma coincidência, este comando executa os comandos associados ao valor coincidente. Se o swicth case não encontra uma coincidência, a execução continua no bloco chamado "default" ou no primeiro comando seguinte ao *switch*.

Entretanto, C nos reserva uma pequena "armadilha" associada com o comando switch case. Durante o teste das coincidências dentro da lista passada pelo programador, se o programa encontrar um valor coincidente, ele executa os comandos associados ao valor e, a menos que esses comandos sejam seguidos da instrução "break", o programa continua sua execução na instrução associada aos outros valores restantes da lista – por exemplo: veja o programa à seguir:

CASE1.CPP

```
1  /* Programa: case1.c
2     Usando swicth case de maneira incorreta */
3  #include <stdio.h>
4  #include <conio.h>
5  void main(){
6    char letra;
7
8    printf("Entre com apenas uma letra: ? ");
9    scanf("%c", &letra);
10
11   switch (letra){
12     case 'A': printf("Vogal A\n");
13     case 'E': printf("Vogal E\n");
14     case 'I': printf("Vogal I\n");
15     case 'O': printf("Vogal O\n");
16     case 'U': printf("Vogal U\n");
17   }
18
19   getch();
20 }
```

Exemplo de saída gerada:

```
Entre com apenas uma letra: ? E
Vogal E
Vogal I
Vogal O
Vogal U
```

Note que apesar da coincidência com a letra "E" o programa continuou a executar os outros case's (linhas 13 em diante) até encontrar o fim do switch (linha 17), ou continuaria assim até encontrar um comando "break" (que faltou). Isto é, uma vez que o programa tenha encontrado um valor que coincida com algum case, ele continua a executar os próximos case's, não importando mais se ocorrem coincidências ou não.

A maneira correta de codificar o programa acima é mostrado à seguir (não esquecer de usar "break" associado ao final de cada "case"):

CASE2.CPP

```
/* Programa: case2.c
   Usando swicth case de maneira correta */
```

```
#include <stdio.h>
#include <conio.h>
void main(){
  char letra;
  printf("Entre com apenas uma letra: ? ");
  scanf("%c", &letra);

  switch (letra){
    case 'A': printf("Vogal A\n"); break;
    case 'E': printf("Vogal E\n"); break;
    case 'I': printf("Vogal I\n"); break;
    case 'O': printf("Vogal O\n"); break;
    case 'U': printf("Vogal U\n"); break;
  }

  getch();
}
```

Desta vez, é gerada uma saída como:

```
Entre com apenas uma letra: ? E
Vogal E
```

Note que nos exemplos anteriores não fizemos uso da condição "default" utilizada para executar um bloco no caso de nenhuma coincidência ter sido encontrada dentro da lista programada.

No caso do programa anterior, poderíamos contemplar o caso do usuário entrar com uma letra que não corresponde à nenhuma das vogais verificadas, neste caso, a letra seria uma consoante. O programa ficaria então:

CASE3.CPP

```
/* Programa: case3.c
   Usando swicth case com condição especial "default" */
#include <stdio.h>
#include <conio.h>
void main(){
  char letra;
  printf("Entre com apenas uma letra: ? ");
  scanf("%c", &letra);

  switch (letra){
    case 'A': printf("Vogal A\n"); break;
    case 'E': printf("Vogal E\n"); break;
    case 'I': printf("Vogal I\n"); break;
    case 'O': printf("Vogal O\n"); break;
    case 'U': printf("Vogal U\n"); break;
    default:  printf("Foi digitada uma consoante\n");
  }

  getch();
}
```

Exemplo de saída gerada:

```
Entre com apenas uma letra: ? d
Foi digitada uma consoante
```

Observação importante:



Note que o comando switch..case só trabalha com variáveis do tipo int ou char!

Nos casos anteriores, o que aconteceria se o usuário tivesse entrado com uma vogal minúscula ao invés de uma das maiúsculas programas para testar as vogais??? Como resolver este problema???

Note que o comando swicth..case é bastante flexível e permite testar faixas de valores ou combinações de condições. Neste caso, devemos usar a vírgula (",") para separar as coincidências que correspondem à um mesmo case.

A solução para o caso do usuário entrar com uma vogal minúscula ficaria então como mostrado abaixo:

CASE4.CPP

```
/* Programa: case4.c
   Usando swicth case de forma mais flexivel */
#include <stdio.h>
#include <conio.h>
void main(){
  char letra;
  printf("Entre com apenas uma letra: ? ");
  scanf("%c", &letra);

  switch (letra){
    case 'A', 'a': printf("Vogal A\n"); break;
    case 'E', 'e': printf("Vogal E\n"); break;
    case 'I', 'i': printf("Vogal I\n"); break;
    case 'O', 'o': printf("Vogal O\n"); break;
    case 'U', 'u': printf("Vogal U\n"); break;
    default:  printf("Foi digitada uma consoante\n");
  }

  getch();
}
```

Que gera uma saída do tipo mostrada abaixo:

```
Entre com apenas uma letra: ? i
Vogal I
```

Capítulo 4

Estruturas de Repetição

Neste capítulo serão estudados os comandos:

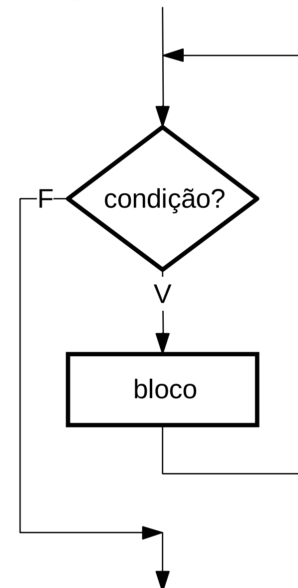
- while (*condição*){*bloco de comandos*}
- do {*bloco de comandos*} while (*codição*);
- for (*inicialização*; *condição*; *incremento*) {*bloco de comandos*}

Contents

4.1 Repetição com teste no início: while

Esta estrutura permite repetir um bloco de programação enquanto a condição testada for verdadeira. Caso contrário, o programa sai deste bloco de repetição e continua sua execução pelas linhas seguintes de código.

Fluxograma:



Sintaxe:

while (*condição*) {
/ bloco de comandos à ser repetido*
*enquanto a condição for válida */*
}

Exemplo$_1$: Suponha que queremos montar um programa que continue repetindo um teste (como o do programa CASE4.CPP, pág 55), se ao final o usuário digitar "S" (Sim).

WHILE1.CPP

```
1  /* Programa: while1.cpp
2     Usando while */
3  #include <stdio.h>
4  #include <conio.h>
5  void main(){
6    char letra;
7    char continua='S';
8
9    while (continua=='S'||continua=='s') {
10     /***************************************************************/
11     clrscr();
12     printf("Entre com apenas uma letra: ? ");
13     fflush(stdin);  /* limpa buffer de teclado */
14     scanf("%c", &letra);
15     switch (letra){
16       case 'A':
17       case 'a': printf("Vogal A\n"); break;
18       case 'E':
19       case 'e': printf("Vogal E\n"); break;
20       case 'I':
21       case 'i': printf("Vogal I\n"); break;
22       case 'O':
23       case 'o': printf("Vogal O\n"); break;
24       case 'U':
25       case 'u': printf("Vogal U\n"); break;
26       default: printf("NÆo foi informada nenhuma vogal.\n");
27     }
28     /***************************************************************/
29     printf("\nDigite S para continuar no programa: ");
30     fflush(stdin);  /* limpa buffer de teclado */
31     continua = getchar();
32     // scanf("%c", &continua);
33     // printf("continua = %c\n", continua);
34   } /* fecha o while */
35   printf("Programa abortado\n");
36   getch();
37 }
```

Note que na linha 9 estabelecemos 2 condições para que o programa execute o teste que queremos repetir (linhas 11 à 27) até que o usuário aperte a tecla "S" ou "s". Note que como a condição já é testada no início

do bloco while, se faz necessário "forçar" uma condição inicial para que este teste seja validado ao menos uma única vez e assim o teste seja executado ao menos uma vez. Por isto, declaramos a variável *continua* e a inicializamos já diretamente com 'S' – linha 7.

Note ainda que aprimoramos um pouco o programa CASE4.CPP da página 55, de forma que o mesmo identifique os casos em que o usuário digitou uma vogal minúscula ou maiúscula. Para tanto, repare que para testar se o usuário digitou 'A' ou 'a', foram usados 2 case's, um para cada letra, mas entre o primeiro case e o segundo, propositalmente não foi utilizado o comando break.

Foram utilizadas as linhas contendo as instruções:

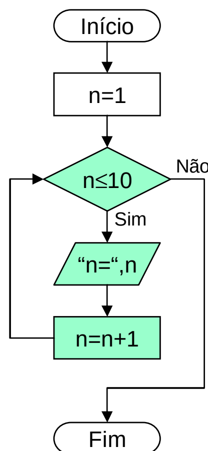
```
fflush(stdin);
```

para evitar erros de execução que ocorrem pelo fato do programa não ter ler 2 seqüencias de entrada de dados pelo teclado. O erro pode ocorrer porque o programa não esvazia corretamente o "*buffer*" do teclado e acaba passando para o segundo comando de leitura do teclado o próprio caracter correspondente à tecla "Return" ou "Enter" (código 13 na tabelas ASCII) usado pelo usuário para entrar com o primeiro dado e assim, o programa recebe um conteúdo vazio (string nula) para o segundo comando de entrada de dados.

Exemplo$_2$: O programa à seguir executa repetidas o bloco presente dentro do while até que a variável n alcance um valor igual ou inferior à 10.

Solução:

a) Fluxograma:



b) Programa em C:

WHILE2.CPP
```
#include <stdio.h>
void main(){
  int n=1;

  while (n <= 10) {
    printf("n= %2i\n", n);
    n = n + 1;
  }
}
```

c) Saída do programa:

```
n=  1
n=  2
n=  3
n=  4
n=  5
n=  6
n=  7
n=  8
n=  9
n= 10
```

Problema$_1$: Modifique o programa anterior (WHILE2.CPP) para que n comece com $n = 10$ e termine com $n = 1$.

Solução:

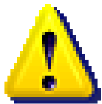
WHILE2.CPP
```
#include <stdio.h>
#include <conio.h>
void main(){
    int n=10;

    while(n>=1){
        printf("n= %2i\n", n);
        n=n-1;
  }
}
```

Saída do programa:

```
n= 10
n=  9
n=  8
n=  7
n=  6
n=  5
n=  4
n=  3
n=  2
n=  1
```





Note:
toda estrutura de repetição deve possuir uma **condição de parada**. Quando isso não acontece, ocorre o que chamamos de "**looping infinito**".

Problema$_2$: E se fosse desejada uma saída como a mostrada abaixo?

```
  n |  t
----+----
 10 |  1
  9 |  2
  8 |  3
  7 |  4
  6 |  5
  5 |  6
  4 |  7
  3 |  8
  2 |  9
  1 | 10
```

Solução:

WHILE03.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
   int n=10;

   printf("  n |  t\n");
   printf("----+----\n");
   while(n>=1){
      printf(" %2i | %2i\n", n, (10-n)+1);
      n=n-1;
   }
}
```

Problema$_3$: Escreva um programa que gere uma tabela de conversão de graus Fahrenheit para graus Célsius, iniciando em -10°F até +80°F, avançando de 5 em 5° graus Fahrenheit, baseado na equação abaixo:

$$^oC = \frac{5(^oF - 32)}{9} \tag{4.1}$$

Solução:

WHILEFAR.CPP
```
1  #include <stdio.h>
2  #include <conio.h>
3  void main(){
4     int f=-10;
5     float c;
6
7     clrscr();
8     printf(" oF |    oC\n");
9     printf("----+-------\n");
10    //      1234  123456
11
12    while( f<=80 )
13    {
14       c=(5.0*(f-32))/9.0;
15       printf("%3i | %6.1f\n", f, c);
16       f= f+5;
17    }
18 }
```

Saída:

```
 oF |    oC
----+-------
-10 |  -23.3
 -5 |  -20.6
  0 |  -17.8
  5 |  -15.0
 10 |  -12.2
 15 |   -9.4
 20 |   -6.7
 25 |   -3.9
 30 |   -1.1
 35 |    1.7
 40 |    4.4
 45 |    7.2
 50 |   10.0
 55 |   12.8
 60 |   15.6
 65 |   18.3
 70 |   21.1
 75 |   23.9
 80 |   26.7
```

Atenção:



No programa anterior, foram declaradas 2 tipos diferentes de variáveis: `int f` para graus Fahrenheit e `float c` para graus Célsius. Como a temperatura em graus Fahrenheit só varia em torno de valores inteiros, esta variável foi declarada como `int`. Mas note que usando a equação 4.5, é inevitável o surgimento de casas decimais na conversão de graus Fahrenheit para graus Célsius pela divisão sendo realizada. Note porém que para codificar corretamente a equação 4.5 devem ser acrescentados `.0` nas constantes 5 e 9 desta equação (ver linha 14 no programa WHILEFAR.CPP), para forçar o C a realizar o cálculo da divisão em ponto flutuante (com casas decimais). Caso contrário o C truncará a parte correspondente às casas decimais, e o resultado (incorreto) será uma coluna de graus Célsius com a casa decimal zerada.

Problema$_4$**:** Escreva um programa que calcule o fatorial do número n digitado pelo usuário.

Embasamento teórico: em teoria, o seguinte cálculo deveria ser ralizado, por exemplo:
Fatorial de 5: $5! = 5 \times 4 \times 3 \times 2 \times 1$. Levando-se ainda em conta que, por definição: $0! = 1$ e que não existem o fatorial de números negativos.

Solução$_1$**:** segue abaixo uma primeira versão para este programa:

FAT01.CPP
```
#include <stdio.h>
#include <conio.h>
void main(){
  int n, fat;

  printf("\n\nFatorial de: ? "); scanf("%i", &n);
  printf("%i! = ", n);

  if (n<0){
     printf("Não existe (n é negativo)\n");
  }
```

```
12    else{
13       if (n==0){
14          printf("1\n");
15       }
16       else{
17          if (n==1){
18             printf("1\n");
19          }
20          else{
21             fat=n*(n-1);
22             n=n-2;
23             while (n>=2){
24                fat= fat*n;
25                n= n-1;
26             }
27             printf("%i\n", fat);
28          }
29       }
30    }
31 }
```

Note alguns detalhes no programa anterior:

1. As linhas 9 à 11 verificam se é o caso do usuário tentar forçar o programa a calcular o fatorial de um número negativo;

2. Caso contrário, as linhas 13 à 15 verificam ainda se o usuário está tentando obter o fatorial de zero que por definição é 1;

3. Caso então o usuário tenha entrado com $n > 0$ o programa passa a tentar calcular o fatorial entre as linhas 17 e 29.

4. Entretanto, pela maneira peculiar do programa calcular o fatorial de um número (linhas 21 à 28), as linhas 17 à 19 prevêem o caso do usuário ter requisitado o cálculo do fatoral de 1 que é 1.

5. A principal rotina do programa do programa anterior que realmente calcula o fatorial de um número está entre as linhas 21 à 28:

```
21    fat=n*(n-1);
22    n=n-2;
23    while (n>=2){
24       fat= fat*n;
25       n= n-1;
26    }
```

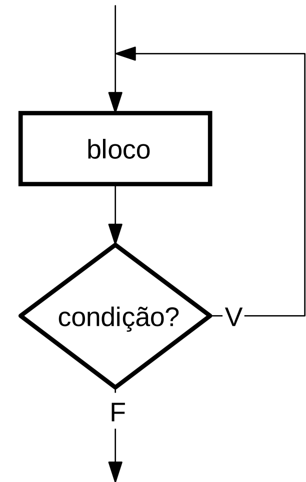
Seja n por exemplo, igual à 5, então, será realizado o seguinte processamento:

21	`fat=n*(n-1);`	$fat = 5 * (5 - 1) = 5 * 4 = 20$
22	`n=n-2;`	$n = 5 - 2 = 3$
23	`while (n>=2){`	$(3 \geq 2)$? – Sim, então entre no while:
24	`fat= fat*n;`	$fat = 20 * 3 = 60$
25	`n= n-1;`	$n = 3 - 1 = 2$
26	`}`	Volte para o while (linha 23)
23	`while (n>=2){`	$(2 \geq 2)$? – Sim, então entre no while:
24	`fat= fat*n;`	$fat = 60 * 2 = 120$
25	`n= n-1;`	$n = 2 - 1 = 1$
26	`}`	Volte para o while (linha 23)
23	`while (n>=2){`	$(1 \geq 2)$? – Não, fim do while, vá para linha 27
27	`printf("%i\n", fat);`	Faz aparecer na tela: "120"

4.2 Repetição com teste no final: do..while

Estrutura de repetição de um bloco de comandos similar ao comando while, mas neste caso, a condição é testada apenas no final do bloco de repetição. Isto significa que o bloco de repetição vai ser executado no mínimo uma vez.

Fluxograma:



Sintaxe:

```
do {
    bloco de comandos à serem repetidos
} while (codição);
```

Exemplo$_1$: Suponha que queremos montar um programa que continue repetindo um teste (como o do programa WHILE1.CPP, pág 55), se ao final o usuário digitar 'S' ou 's' (Sim). Mas agora este teste é executado apenas no final do bloco de repetição.

DOWHILE1.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  char letra;
  char continua;

  clrscr();
  do {
    /***************************************************************/
    printf("Entre com apenas uma letra: ? ");
    fflush(stdin);   /* limpa buffer de teclado */
    scanf("%c", &letra);
    switch (letra){
      case 'A':
      case 'a': printf("Vogal A\n"); break;
      case 'E':
      case 'e': printf("Vogal E\n"); break;
      case 'I':
      case 'i': printf("Vogal I\n"); break;
      case 'O':
      case 'o': printf("Vogal O\n"); break;
      case 'U':
      case 'u': printf("Vogal U\n"); break;
      default:  printf("Não foi informada nenhuma vogal.\n");
    }
    /***************************************************************/
    printf("\nDigite S para continuar no programa: ");
    fflush(stdin);   /* limpa buffer de teclado */
    continua = getchar();
    // scanf("%c", &continua);
    // printf("continua = %c\n", continua);
  } while (continua=='S'||continua=='s'); /* fecha o do..while */
  printf("Programa abortado\n");
  getch();
```

```
}
```

Saída do programa

```
Entre com apenas uma letra: ? a
Vogal A

Digite S para continuar no programa: s
Entre com apenas uma letra: ? E
Vogal E

Digite S para continuar no programa: S

Entre com apenas uma letra: ? n
Não foi informada nenhuma vogal.

Digite S para continuar no programa: n

Programa abortado
```

Exemplo$_2$: Programação de um menu.

MENU.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
   char opcao;

   do {
      clrscr();

      do {
         printf("\n\nMenu:\n");
         printf("1. Inclusão\n");
         printf("2. Alteração\n");
         printf("3. Exclusão\n\n");
         printf("0. Sair do programa\n");
         printf("\nDigite uma opção: ? ");
         fflush(stdin);  /* limpa buffer de teclado */
         scanf("%c", &opcao);
         switch (opcao){
           case '1': printf("\nFoi escolhido Inclusão\n"); break;
           case '2': printf("\nFoi escolhido Alteração\n"); break;
           case '3': printf("\nFoi escolhido Exclusão\n"); break;
           //case '0': printf("Foi escolhido Sair do programa\n"); break;
           default: printf("\nNão foi digitada nenhuma opção válida!");
         }
      } while (opcao!='1' && opcao!='2' && opcao!='3' && opcao!='0');

      printf("\nContinuar no programa (S/N): ? ");
      fflush(stdin);  /* limpa buffer de teclado */
      scanf("%c", &opcao);

   } while (opcao!='N' && opcao!='n');

   printf("Ok... saindo do programa\n");
   getch();
}
```

Saída do programa:

```
Menu:
1. Inclusão
2. Alteração
```

```
3. Exclusão

0. Sair do programa

Digite uma opção: ? 1

Foi escolhido Inclusão

Continuar no programa (S/N): ? s


Menu:
1. Inclusão
2. Alteração
3. Exclusão

0. Sair do programa

Digite uma opção: ? g

Não foi digitada nenhuma opção valida!

Menu:
1. Inclusão
2. Alteração
3. Exclusão

0. Sair do programa

Digite uma opção: ? 1

Foi escolhido Inclusão

Continuar no programa (S/N): ? n
Ok... saindo do programa
```

Problema$_2$: Como você modificaria o programa anterior para que o menu de operação do programa trabalhasse da forma mostrada abaixo?

```
Menu:
1. Inclusão
2. Alteração
3. Exclusão
0. Sair do programa

Digite uma opção: ? 1

Foi escolhido Inclusão

Menu:
1. Inclusão
2. Alteração
3. Exclusão
0. Sair do programa

Digite uma opção: ? g

Não foi digitada nenhuma opção valida!

Menu:
1. Inclusão
2. Alteração
3. Exclusão
0. Sair do programa
```

```
Digite uma opção: ? 0

Confirma saída do programa: ? s
Ok... Saindo do programa
```

Problema$_3$: Monte um programa capaz de determinar o valor do **somatório** final, h, atingido para a **série** mostrada abaixo conforme o valor n digitado pelo usuário.

$$h = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} \tag{4.2}$$

Solução: Para entender a lógica do algoritmo que deve ser construído é melhor simular antes o cálculo de alguns valores de h conforme o valor de n digitado pelo usuário, o que resulta numa tabela como a mostrada à seguir:

n	h	*Repare*:
1	$h = 1$	$h_1 \leftarrow 1$
2	$h = 1 + 1/2 = 1,5$	$h_2 \leftarrow h_1 + 1/2$
3	$h = 1 + 1/2 + 1/3 = 1,8\dot{3}$	$h_3 \leftarrow h_2 + 1/3$
4	$h = 1 + 1/2 + 1/3 + 1/4 = 2,08\dot{3}$	$h_4 \leftarrow h_3 + 1/4$
$\vdots$	$\vdots$	$\vdots$
n	h=1+1/2+1/3+...+1/n	$h_n \leftarrow h_{n-1} + 1/n$

Tabela 4.1: Simulando valores de $h \times n$.

Reparamos pela tabela anterior, que entre uma interação e outra ocorre o seguinte tipo de cálculo:

$$h_n \leftarrow h_{n-1} + 1/n \tag{4.3}$$

onde h_n pode ser interpretado como o valor atual (ou novo valor) sendo buscado e h_{n-1} pode ser interpretado como o valor anterior. Percebemos que h sempre pode iniciar com 1 e que conforme n aumenta, novas interações para o cálculo de h são necessárias conforme a equação 4.3. Notamos entretanto que é necessário se controlar o número de vezes em que o cálculo de h deve ser repetido conforme demonstra a tabela 4.1 levantada anteriormente. Necessitamos então de uma variável *contadora* para controlar o número de vezes que a equação 4.3 deve ser repetida. Percebemos pela tabela 4.1, que esta variável *contadora* deve iniciar em 1 e terminar em n conforme especificado pelo usuário, e que h deve ser inicializado com um valor nulo.

Programa:

SERIE.CPP

```
/* Programa: calculo do somatorio de uma serie convergente...
   h = 1 + 1/2 + 1/3 + 1/4 + ... + 1/n
*/
#include <stdio.h>
#include <conio.h>
void main(){
  int n,cont;
  float h;

  printf("Entre com n: ? "); scanf("%i", &n);

  h=0.0; cont=1;
  do{
    printf("%d | h=%f+1/%d=", cont, h, cont);
    h=h+1.0/cont;
    printf("%f\n", h);
    cont=cont+1;
  } while( cont<=n );

  printf("\nValor final de h: %f\n", h);
}
```

Obs.: Note que as linhas 14 e 16 não são necessárias para que o algoritmo funcione, apenas estão presentes para mostrar na tela, como o valor de h está sendo evoluído.

Saída do Programa:

```
Entre com n: ? 5
1 | h=0.000000+1/1=1.000000
2 | h=1.000000+1/2=1.500000
3 | h=1.500000+1/3=1.833333
4 | h=1.833333+1/4=2.083333
5 | h=2.083333+1/5=2.283334

Valor final de h: 2.283334
```

Problema Proposto: Melhore o programa para cálculo do fatorial (em comparação ao programa da pág. 61).

4.3 Uso de “*flags*” para controle de programas

Quando não sabemos à priori o número de repetições que devem ser executadas para um certo bloco do programa, podem ser criadas variáveis Booleanas (Verdadeiro ou Falso) para verificar se certas condições que caracterizam o fim do laço de repetição foram alcançadas. Estas variáveis apenas sinalizariam este tipo de condições e por isto mesmo são comumente chamadas de “***flags***”.

Exemplo$_1$: Escreva um programa que imprima: a) o menor, b) o maior e c) o valor médio de um conjunto de dados digitados pelo usuário. Detalhe: repare que o que determina o final da entrada de dados é a digitação de um número negativo. Note então que a condição de saída do laço de repetição será a leitura de um valor negativo – este seria o “*flag*” neste caso.

MEDIAIT.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  float valor, soma, maior, menor, media;
  int cont=0;
  soma=0; // inicializa variável que contem o somatório dos números entrados

  printf("Programa para gerar a média dos valores informados\n");
  printf("Entre com um número negativo para indicar o fim da entrada de dados\n\n");

  do {
     printf("No. %2i) ? ", (cont+1));
     scanf("%f", &valor);
     if (valor>=0){ // somente realiza cálculos se valor entrado >= zero
        cont=cont+1;
        if (cont==1){
           maior=valor; // supõe que o maior valor é o primeiro informado
           menor=valor; // supõe que o menor valor é o primeiro informado
        }
        soma=soma+valor;
        if (valor>maior)
           maior=valor;
        if (valor<menor)
           menor=valor;
     }
  } while (valor >= 0);
```

```

  if (cont>0){ // usuário entrou com ao menos 1 número >= zero
     media=soma/cont;
     printf("\nForam informados %i números...\n", cont);
     printf("\n      Média: %7.2f\n", media);
     printf("Menor valor: %7.2f\n", menor);
     printf("Maior valor: %7.2f\n", maior);
  }
  else {
     printf("Não foi informado nenhum número maior ou igual à zero\n");
     printf("Programa abortado\n");
  }
  printf("\nAperte uma tecla para sair do programa ");
  getch();
}
```

Saída gerada:

```
Programa para gerar a média dos valores informados Entre com um
número negativo para indicar o fim da entrada de dados

No.  1) ? 10
No.  2) ? 5
No.  3) ? -9

Foram informados 2 números...

      Média:    7.50
Menor valor:    5.00
Maior valor:   10.00

Aperte uma tecla para sair do programa
```

4.4 Algoritmos Interativos

Para resolver problemas científicos, **métodos interativos** são freqüentemente empregados. Num método interativo, uma primeira aproximação para uma solução é obtida e então algum método que melhore a precisão do resultado é repetido até que duas aproximações sucessivas alcancem a precisão desejada. O processo pode ser descrito como:

```
Produza a primeira aproximação na variável old;
Produza uma aproximação melhor com base no valor
        da variável old e guarde este novo resultado na variável new;
while old e new não estejam próximos o suficiente
{
    old = new;
    Produza uma aproximação melhor com base no valor
            da variável old e guarde este novo resultado na variável new;
}
```

Exemplo: Programa para determinar a raiz quadrada de um número.
Se *old* é a primeira aproximação para a raiz quadrada do número x, então, a melhor aproximação, *new*, pode ser encontrada fazendo-se:

$$new = (old + \frac{x}{old})/2 \tag{4.4}$$

Por exemplo, supondo que 3 seja uma aproximação para a raiz quadrada do número 10[1], aplicando-se a equação 4.4 acima umas 3 vezes resulta em:

old	*new*
3	(3+10/3)/2 = 3.17
3.17	(3.17+10/3.17)/2 = 3.1623
3.1623	(3.1623+10/3.1623)/2 = 3.162278

Note que esta seqüencia rapidamente converge para o valor real ($\sqrt{10}$ = 3.1622776601683793319988935444327). Isto pode não acontecer para todos os algoritmos interativos (eles podem "convergir" ou "divergir"). Mas neste caso, este método sempre converge desde que a primeira aproximação seja um número positivo diferente de zero.

Na explicação genérica para um algoritmo interativo, foi utilizada a expressão "**while** *old* **e** *new* não estejam próximos o suficiente". Como aplicar esta abordagem neste caso? Neste caso, podemos estabelecer que a última nova aproximação é aceitável quando a diferença entre o valor anterior (*old*) e o último valor (*new*) for menor que 0.0005. Isto significa que a estimativa para a raiz quadrada de um número estará correta para as 3 primeiras casas decimais. Então poderia ser realizado um teste como:

```
while ((new - old) > 0.0005) {
    ..
}
```

Entretanto, notar que se for aplicado o teste acima para o caso da estimativa da raiz quadrada do número 10, na 2ª interação será testado: $3.1623 - 3.17 > 0.0005$, o que implica em testar se: $-0.0077 > 0.0005$ – o que infelizmente fará com que a busca por uma nova aproximação pare abruptamente (o algoritmo não continua dentro do while buscando por uma aproximação melhor). A solução para este problema é simples: basta testar se o módulo da diferença entre o novo valor e o valor antigo é menor que 0.0005. Isto pode ser feito aplicando-se a função fabs(*x*) definida na biblioteca <math.h>, que retorna x no caso de x ser ≥ 0 ou $-x$ no caso de x ser negativo. Seria o equivalente a realizar o seguinte teste: $|(new - old)| > 0.0005$?, que em C ficaria então como:

```
while ( fabs(new - old) > 0.0005) {
    ..
}
```

Entretanto ainda existe um outro problema. Em geral tentar convergir 2 valores dentro de uma certa tolerância calculando-se simplesmente a diferença entre estes 2 valores, resulta não no cálculo da tolerância mas sim na diferença absoluta entre os 2 valores (na escala original dos valores empregados. Como se determinar estão esta tolerância? Considere o caso abaixo:

Valor Exato	Aproximação
100	100.1
0.1	0.2

Para o caso acima, qual das 2 aproximações acima é a melhor? As duas resultam num erro absoluto de 0,1, porém note que num dos casos, o erro está em torno de 0,1% das grandezas sendo aproximadas, enquanto no outro caso (o segundo), este erro é de 100%! Note então que precisamos "escalonar" os valores sendo aproximados senão vamos acabar produzindo um método interativo muito ruim se o caso de entrada for similar ao segundo caso mostrado na tabela anterior. Desta forma, uma maneira de avaliar o erro relativo é fazer-se então:

$$erro = \frac{|new - old|}{new}$$

que com certeza nos fará realizar aproximações na escala de porcentagem dos valores sendo tratados e não como aproximações com base em diferenças absolutas de 2 valores sendo tratados. Então finalmente nosso teste fica como:

[1] $\sqrt{10} = 3.1622776601683793319988935444327$

```
while ( (fabs(new - old))/new ) > 0.0005) {
    ..
}
```

Para tornar nosso algoritmo a prova de falhas precisamos nos assegurar de que o usuário não entre com valores negativos ou nulos. No caso do valor ser nulo (zero) basta informar ao usuário que: $\sqrt{0} = 0$ – assim, este configuraria um "caso especial", fácil de ser tratado. Para o caso do usuário ter entrado com um número negativo, o programa simplesmente indica que não calcula a raiz quadrada de números negativos e para de ser executado. E por fim, como aproximação inicial para a raiz quadrada de um número, vamos simplesmente fazer com que: $\sqrt{x} = ?$; $old = x$ e usamos a equação 4.4 para encontrar um novo valor (*new*) até que uma tolerância relativa de 0.000005 seja encontrada (o que resulta num erro de 0.05%).

Programa:

SQUARE1.CPP

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(){
  float tol, valor, old, new_v, erro;
  /* 'new' não pode ser usado pois confunde o
  compilador: existe um operador chamado 'new' usado para definir um novo
  objeto - caso de programação orientada a objetos */

  int cont=0; // apenas para contar o numero de interações
  tol=0.000005;

  clrscr();
  printf("Programa para extrair a raiz quadrada de um numero\n\n");
  printf("Entre com o numero: ? ");
  scanf("%f", &valor);

  if (valor < 0.0){
      printf("Este programa não pode encontrar a raiz quadrada\n");
      printf("de números negativos.\n");
  }
  else{
      if (valor == 0){
          printf("A raiz quadrada de 0 é: 0.0\n");
      }
      else {
          // realizando a primeira aproximação
          old=valor;
          new_v=(old+valor/old)/2.0;
          erro=(fabs(new_v-old))/new_v;
          //                  1234567890   1234567890   123456789012
          printf("Interação |    Old      |    New      |  Tolerância \n");
          printf("    1      | %10.6f | %10.6f | %12.10f\n", old, new_v, erro);
          cont=cont+1;

          while (erro > tol) {
             cont=cont+1;
             old = new_v;
             new_v=(old+valor/old)/2.0;
             erro=(fabs(new_v-old))/new_v;
             //                  123456789   123456789   12345678901
             printf("   %2i      | %10.6f | %10.6f | %12.10f\n", cont,
                         old, new_v, erro);
          }

          printf("\n A raiz quadrada de %f é: %f\n", valor, new_v);
      }
  }
  printf("\nAperte uma tecla para sair do programa ");
  getch();
}
```

Saída gerada:

```
Programa para extrair a raiz quadrada de um numero

Entre com o numero: ? 10

Interação |     Old      |     New      | Tolerância
    1     |   10.000000 |    5.500000 | 0.8181818128
    2     |    5.500000 |    3.659091 | 0.5031055808
    3     |    3.659091 |    3.196005 | 0.1448952258
    4     |    3.196005 |    3.162456 | 0.0106087020
    5     |    3.162456 |    3.162278 | 0.0000562444
    6     |    3.162278 |    3.162278 | 0.0000000000

 A raiz quadrada de 10.000000 é: 3.162278

Aperte uma tecla para sair do programa
```

4.5 Repetição Automática: for

Esta estrutura permite repetir um bloco de programação conforme varia o conteúdo de uma variável contadora usada para controlar o laço de repetição. Note que diferente do while ou do..while, aqui o programador é obrigado à saber de antemão o número de vezes que o bloco contido dentro do laço for deve ser repetido.

Sintaxe:

```
for ( var_contadora=valor_inicial ; condição_de_parada ; atualização_var_contadora ) {
    /* bloco de comandos à ser repetido de acordo com var_contadora */
}
```

Note que o comando for é composto internamente por 3 expressões:

```
for ( expressão_1; expressão_2; expressão_3 )
{
    /* Bloco à ser repetido */
}
```

onde: *expressão_1* especifica a inicialização do laço de repetição;
expressão_2 especifica um teste feito **antes** de cada interação, e;
expressão_3 especifica um incremento ou decremento que é realizado após cada iteração.
O laço termina conforme o teste executado pela *expressão_2*.

Note que este comando é equivalente à fazer:

```
expressão_1;
while ( expressão_2 ) {
    /* Bloco à ser repetido */
    expressão_3;
}
```

Exemplos

Exemplo$_1$: Programa simples mostrando modo de operação do laço for. Incrementando e decrementando uma variável contadora.

FOR1.CPP
```
1  /* Programa para demonstrar uso do FOR */
2  #include <stdio.h>
3  #include <conio.h>
4  void main(){
5    int cont;
6    clrscr();
7
8    // incrementando variavel contadora...
9    for(cont=1; cont<=10; cont=cont+1){
10     printf("%i, ", cont);
11   }
12   printf("\n\n");
13
14   // decrementando variavel contadora...
15   for(cont=10; cont>=1; cont--){
16     printf("%i, ", cont);
17   }
18   printf("\n\n");
19
20   // incrementado variavel contadora com passo qualquer...
21   for(cont=1; cont<=10; cont=cont+3){
22     printf("%i, ", cont);
23   }
24   printf("\n\n");
25 }
```

O programa acima gera a seguinte saída:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,

1, 4, 7, 10,
```

Note que na linha 3 do programa anterior, foi usado um comando de decremento disponível apenas na linguagem C:

Acronismo de C	Equivalente à fazer:
cont-;	cont=cont-1;
cont++;	cont=cont+1;

O fluxograma correspondente ao laço de repetição for aparece na figura 4.1 à seguir.

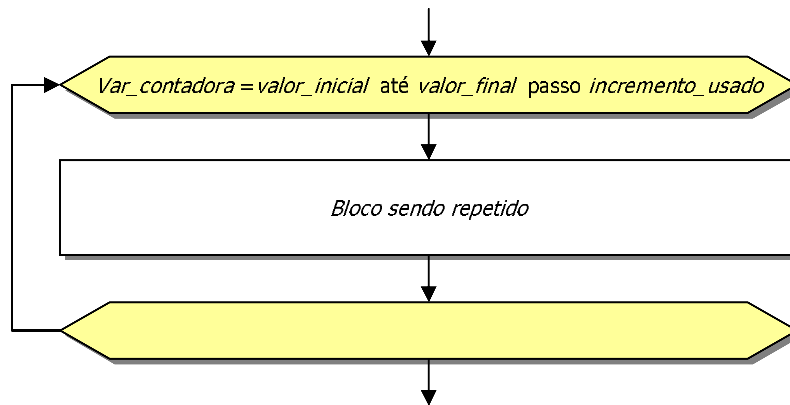


Figura 4.1: Fluxograma correspondente ao laço for.

Exemplo$_2$: Monte um Programa para gerar uma tabela de conversão de graus Fahrenheit para graus Célsius, iniciando em -10°F até +80°F, avançando de 5 em 5° graus Fahrenheit, baseado na equação abaixo:

$$^oC = \frac{5(^oF - 32)}{9} \tag{4.5}$$

Algoritmo em C:

FORTEMP.CPP

```
/* 2o Programa para demonstrar uso do FOR */
#include <stdio.h>
#include <conio.h>
void main(){
  int fare; // note: o nome "far" e' palavra reservada do Builder C++
  float celsius;
  clrscr();

  printf(" Fahrenheit | Celsius\n");
  printf("------------+-----------\n");
  /*          100      |   -67.46
          123412312345|12123456123 */
  for (fare=-10; fare<=80; fare=fare+5){
      celsius=(5.0*(fare-32))/9.0;
      printf("    %3i     |   %6.2f\n", fare, celsius);
  }
}
```

Saída gerada:

```
 Fahrenheit | Celsius
------------+-----------
    -10     |  -23.33
     -5     |  -20.56
      0     |  -17.78
      5     |  -15.00
     10     |  -12.22
     15     |   -9.44
     20     |   -6.67
     25     |   -3.89
     30     |   -1.11
     35     |    1.67
     40     |    4.44
     45     |    7.22
     50     |   10.00
     55     |   12.78
     60     |   15.56
     65     |   18.33
     70     |   21.11
     75     |   23.89
     80     |   26.67
```

Compare a solução deste problema com o mesmo já realizado na página 61, mas usando while.

Exemplo$_3$: Monte um programa que determina o fatorial no número n digitado pelo usuário, mas usando for.
Solução:

FATFOR.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int n,i;
  int fat=1;

  clrscr();
  printf("Fatorial de ? ");
  scanf("%i", &n);

  for (i=n;i>=2;i--){
    fat=fat*i;
  }

  printf("%i!= %i\n", n, fat);
}
```

Saída gerada:

```
Fatorial de ? 5
5!= 120
```

Exemplo$_4$: Monte um programa que gere n colunas com quantidades aleatórias de caracteres *'s (entre 1 à 76 colunas). O valor de n é especificado pelo usuário.

Dica: ....

Use a função: int random (int num); do C para gerar números aleatórios entre 0 e $num - 1$. Note que é interessante antes de rodar a função random(), preparar o gerador de números aleatórios chamando a função: void randomize(void); – este comando necessita ser executado apenas uma única vez antes do comando random(). Veja o programa exemplo a seguir que imprime uma lista de 10 números aleatórios sorteados entre 0 à 99:

NUMALE.CPP

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // contem randomize() e random() - num aleatorios...
void main(){
  int i;     // contador
  int num;   // numero aleatorio
  clrscr();
  randomize();
  for (i=1; i<=10; i++){
    num=random(100);
    printf("%2i) num= %i\n", i, num);
  }
}
```

Saída gerada:

```
 1) num= 89
 2) num= 28
 3) num= 89
 4) num= 25
 5) num= 92
 6) num= 73
 7) num= 33
 8) num= 46
 9) num= 93
10) num= 74
```

Solução (para o Exemplo$_3$):

COLALE.CPP

```
1 #include <stdio.h>
2 #include <conio.h>
3 #include <stdlib.h> // contem randomize() e random()
4 void main(){
5   int i,n,cols,j;
6
7   clrscr();
8   randomize();
9   printf("Entre com n: ? ");
10  scanf("%i", &n);
11
12  for(i=1; i<=n; i++){
13    cols=random(77);
14    printf("%2i) ", i);
15    for(j=1; j<=cols; j++){
16      printf("*");
17    }
18    printf("\n");
19  }
20 }
```

Saída do programa:

```
Entre com n: ? 5
 1) ****************
 2) ********************************
 3) **********************************
 4) **************************************************************
 5) ****************************
```

Observação: Note que usamos 2 × **for encadeados**. O primeiro, da linha 12, traz dentro de si o segundo for (mais interno), na linha 15. Não há problemas em incluir um for dentro de outro. Nestes casos, o programa termina de executar primeiro o laço de repetição que corresponde ao for mais "interno" e depois termina de executar o laço de repetição do for mais "externo". No caso do programa anterior, o programa vai terminar primeiro de executar o for da linha 15 e depois termina de executar o for da linha 12. Como o for da linha 15 está contido dentro do for da linha 12, o for da linha 15 vai acabar sendo repetido n vezes (conforme programado pelo for da linha 12). Justamente por este fato, estes for trabalham com variáveis contadores que controlam o laço, diferentes. O for da linha 12 trabalha com a variável contadora i e o for da linha 15, com a variável contadora j. Não podemos usar a mesma variável contadora para estes dois for, por exemplo, apenas a variável i, porque senão, o for mais interno (da linha 15), vai acabar modificando a própria variável i que também seria usada pelo for mais externo (linha 12) e assim, o for mais externo iria perder o controle, uma vez que o conteúdo da variável i seria modificado a todo instante que rodasse o for mais interno. Portanto, os dois for trabalham com variáveis contadoras diferentes.

A figura 4.2 à seguir traz um fluxograma que pretende esclarecer este caso.

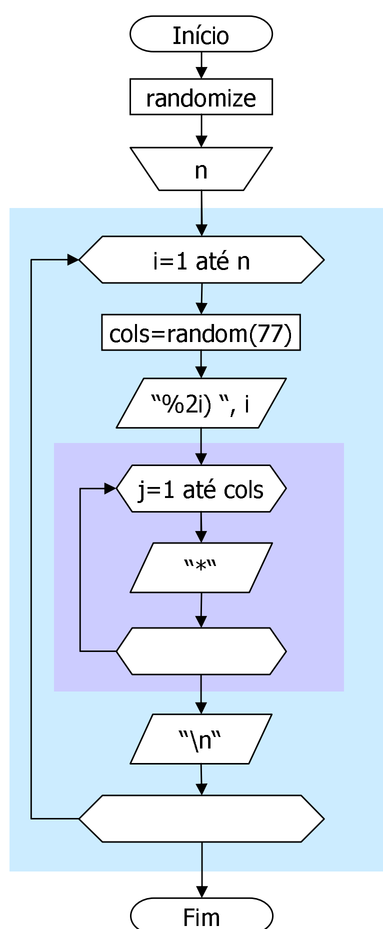


Figura 4.2: Fluxograma para o programa COLALE.CPP.



Note que C permite laços de repetição com mais de uma variável contadora dentro do laço, por exemplo:

FORDUPLO.CPP
```
/* Demonstrando uso do FOR "duplo" */
#include <stdio.h>
#include <conio.h>
void main(){
  char upper, lower;
  clrscr();

  printf("Maiuscula\tMinuscula\n");
  for (lower='a', upper='A'; upper <= 'F'; upper++, lower++){
    printf("%c\t\t%c\n", upper, lower);
  }
}
```

Que gera uma saída como:

```
Maiuscula        Minuscula
A                a
B                b
C                c
D                d
E                e
F                f
```

Atenção porém para a sintaxe correta das 3 expressões que fazem parte do for!

Note ainda que ⇒

Expressão	Valor	Conversão Realizada (Código ASCII)
'A' + 1	'B'	$65 + 1 \rightarrow 66$
'B' + 1	'C'	$66 + 1 \rightarrow 67$
⋮	⋮	⋮
'a' + 1	'b'	$97 + 1 \rightarrow 98$

4.6 while × for

Tanto for quanto while são laços de repetição, mas com a diferença de que, o laço while continua a ser executado enquanto a condição sendo verificada for verdadeira, não importando o valor de uma variável contadora como o exigido pelo laço for. Tanto o bloco de comandos contido dentro do laço de repetição while ou for, somente serão executados se a condição à ser testada for verdadeira, o que significa que os comandos contidos dentro destes blocos podem nem chegar a serem executados.

Comparando ainda while com do..while notamos que o bloco de comandos contido dentro do laço de repetição do..while sempre será executado pelo menos uma única vez, enquanto que no caso do while, pode acontecer deste bloco nunca ser executado se a condição testada no início do laço while se provar falsa.

Exemplos

Exemplo$_1$: Faça um programa que primeiro gere números aleatórios na faixa entre 1 à 20 mostrando a equivalente quantidade na forma de ⋆'s na tela. O programa deve parar de mostrar as colunas de ⋆'s quando o número sorteado for menor do que 5. O programa deve gerar uma saída similar à

mostrada abaixo:

```
12)   ************
 8)   ********
17)   *****************
 3)   Fim do programa
```

Solução: O fluxograma correspondente a uma solução para este problema aparece na figura 4.3.

Algoritmo em C:

BARRAST.CPP

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // contemrandomize() e random()

void main(){
  int cols,j;

  clrscr();
  randomize();

  do{

    cols=random(20);
    printf("%2i) ", cols);

    if (cols>=5)

       for(j=1; j<=cols; j++){
         printf("*");
       } // fim do for
       printf("\n");

    } // fim do if

  } while(cols>=5);

  printf("Fim do programa\n");

}
```

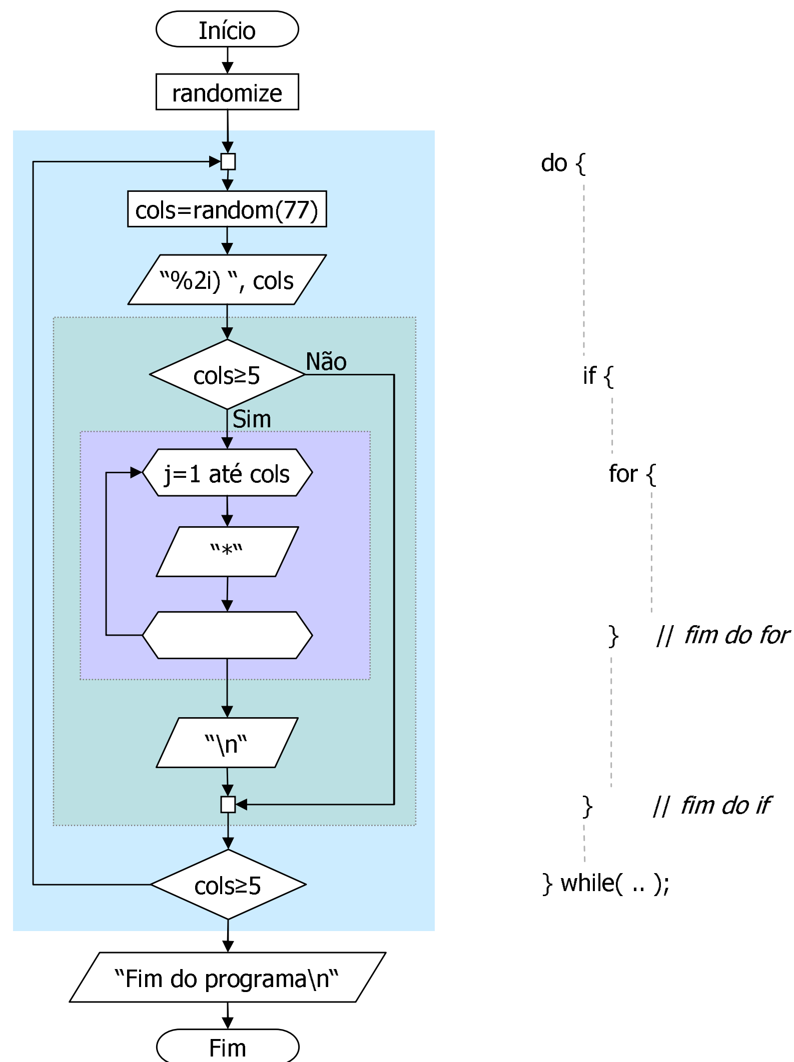


Figura 4.3: Fluxograma solucionando o problema relativo ao exemplo 1 (anterior).

4.7 Instruções break e continue

As instruções vistas anteriormente podem sofrer desvios e interrupções em sua seqüência lógica normal através do uso de break e continue. Entretanto, deve-se **evitar** de utilizar as instruções que veremos a seguir, pois tendem a tornar um programa incompreensível, fogem do que se chama de **programação estruturada**, além de caracterizar uma programação de baixo nível (Assembly, C para alguns microcontroladores).

4.7.1 Instrução break

Esta instrução serve para terminar a execução das instruções de um laço de repetição (for, do...while, while) ou para terminar um conjunto switch...case.

Quando dentro de um laço de repetição, esta instrução força a interrupção do laço independentemente da condição de controle.

Exemplo$_1$ No trecho de programa abaixo, um laço de repetição lê valores para o cálculo de uma média. O laço possui uma condição de controle sempre verdadeira o que, a principio, é um erro: constitui um laço infinito ou *looping* perpétuo. Porém, a saída do laço se dá pela instrução break que é executada quando um valor negativo é lido.

```
printf("Digite valores:");
do{
  printf("Valor: ");
  scanf("%f",&val);
  if(val < 0.0){
    break;      // saída do laço
  }
  num++;
  soma += val; /* faz: soma = soma + val;
}while(1);      // sempre verdadeiro - looping infinito
printf("Média: %f",soma/num);
```

No exemplo anterior, o uso da instrução break deveria ter sido evitada, fazendo-se:

```
printf("Digite valores:");
do{
  printf("Valor: ");
  scanf("%f",&val);
  if(val >= 0.0){
    num++;
    soma += val;
  }
}while(val >= 0.0);
printf("Média: %f",soma/num);
```

Admite-se o uso da instrução break, em estruturas switch...case, porque neste caso serve para separar os conjuntos de instruções em cada case.

Exemplo$_2$: O programa PIANO.CPP abaixo mostra um programa que utiliza a estrutura switch..case com a instrução break para simular um piano no teclado do computador.

PIANO.CPP

```
/**********************************************************************
Programa: PIANO.CPP Proposito: Uso da estrutura switch com break
Ultima Revisao: 27/08/97
**********************************************************************/
#include <dos.h>  // contem funcao sound()
#include <stdio.h>
#include <conio.h>
#define DO  264   // definicao de escala musical
#define RE  297
#define MI  330
#define FA  352
#define SOL 396
```

```
#define LA  440
#define SI  495
void main(){
  int tecla;

  clrscr();
  printf("Digite teclas [z] [x] [c] [v] [b] [n] [m] para notas\n");
  printf("ou [esc] para sair\n");
  do{
    tecla = getch();   // leitura do teclado
    switch(tecla){     // conforme o valor de tecla...
      case 'z':        // se tecla = 'z'
        sound(DO);     // nota do
        break;         // cai fora do switch...
      case 'x':
        sound(RE);
        break;
      case 'c':
        sound(MI);
        break;
      case 'v':
        sound(FA);
        break;
      case 'b':
        sound(SOL);
        break;
      case 'n':
        sound(LA);
        break;
      case 'm':
        sound(SI);
        break;
    }
    delay(200);        // toca por 200 ms
    nosound();         // desliga auto-falante
  }while(tecla != 27); // repete enquanto tecla != [esc]
}
```

4.7.2 Instrução continue

Esta instrução opera de modo semelhante a instrução break dentro de um laço de repetição. Quando executada, ela pula as instruções seguintes do laço de repetição sem entretanto sair do laço. Isto é, a instrução força a avaliação da condição de controle do laço.

Exemplo: No trecho de programa abaixo revemos um laço de repetição que lê valores para o cálculo de uma média. Se (val < 0.0) então o programa salta diretamente para a condição de controle, sem executar o resto das instruções.

```
printf("Digite valores:");
do{
  printf("Valor: ");
  scanf("%f",&val);
  if(val < 0.0){       // se val é negativo...
    continue;          // ...salta para...
  }
  num++;               // se (val < 0.0) estas instruções
  soma += val;         // não são executadas!
}while(val >= 0.0);    // ...fim do laço
printf("média: %f",soma/num);
```

No exemplo anterior, o uso de continue também poderia ter sido evitado fazendo-se:

```
printf("Digite valores:");
do{
```

```
  printf("Valor: ");
  scanf("%f",&val);
  if(val >= 0.0){
    num++;
    soma += val;
  }
}while(val >= 0.0);
printf("Média: %f",soma/num);
```

Resumo

As estruturas de controle são as que definem a lógica de um programa. Fazendo operações como laços (loopings) e if's encadeados, um comportamento complexo pode ser codificado.

Estruturas de Controle	Estruturas que definem a lógica de um programa
if..	Avaliação condicional de um bloco de código
if..else	Escolha entre 2 alternativas e blocos exclusivos de código
while..do	Construção de laço de repetição com condição prévia
do..while	Construção de laço de repetição com teste ao final
for..	Laço de repetição automático e determinístico
break	Saída imediata de um laço interno
continue	Pule diretamente para o teste de condição do laço de repetição

4.8 Problemas Finais

Problema$_1$: Monte um programa que imprima números entre 1 e 20, mas que indique (ao seu lado), se tratar de um número múltiplo de 4.
Por exemplo, uma saída similar à mostrada abaixo deve ser gerada:

```
1...
2...
3...
4...é múltiplo de 4!
5...
6...
7...
8...é múltiplo de 4!
9...
```

Dica: Usar o operador matemático "%" do C para encontrar o resto de uma divisão inteira. Por exemplo: `47 % 2` gera como resultado: `1`, já "`(46%2)`" gera `0`.
Rode o pequeno programa abaixo se continuar com dúvidas:

RESTODIV.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int i;
  clrscr();
  for (i=1; i<=10; i++){
    printf("%i) %i%%3= %i\n", i, i, i%3);
  }
}
```

Problema$_2$: Monte um programa que peça para o usuário entrar com números inteiros positivos, mas o programa deve parar de pedir mais dados de entrada assim que forem digitados 5 números pares. **Dica**: Use novamente o operador "'%" do C.

Problema$_3$: Faça um programa que imprima os números ímpares no intervalo fechado $[a, b]$ (sendo que a e b são definidos pelo usuário).

Problema$_4$: Faça um programa que imprima os N primeiros números da série de Fibonacci: $1, 1, 2, 3, 5, 8, 13, \ldots$. Note que a equação que gera esta série pode ser definida como: $n[i] = n[i-1] + n[i-2]$ para $i \geq 2$ já que $n[0] = n[1] = 1$.

Problema$_5$: Monte um programa que peça para o usuário adivinhar um número escolhido aleatoriamente entre 1 e 100. Se o usuário digitar um número errado, o programa responde o novo intervalo do número procurado. Se o usuário acertou o número procurado, o programa diz quantos palpites foram dados. O programa deve permitir que o usuário faça no máximo 3 tentativas e então deve perguntar se o usuário prefere continuar no jogo. Caso positivo um novo número aleatório é gerado. Caso contrário, o programa é terminado. Por exemplo:

```
O número procurado está entre 1 e 100:
Palpite: ? 45
O número procurado está entre 1 e 44:
Palpite: ? 27
O número procurado está entre 28 e 44:
Palpite: ? 36
Parabéns! Você acertou o número em 3 tentativas.
```

Soluções para os Problemas Propostos

Seguem soluções para os problemas anteriores.

Solução problema$_1$: Números múltiplos de 4 na faixa de 1 à 20.

FOR4.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int i;
  clrscr();
  for (i=1; i<=20; i++){
    printf("%i..", i);
    if(i%4==0){
      printf("é multiplo de 4..");
    }
    printf("\n");
  }
}
```

Solução problema$_2$: Digitação de até 5 números pares.

PROB2.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int val, cont=1, pares=0;
  clrscr();
  do{
    printf("Entre com o numero %i) ", cont);
    scanf("%i", &val);
    if(val%2==0){
      pares++;
      printf("Numeros pares entrados: %i\n", pares);
    }
    cont++;
  } while(pares<5);
}
```

Solução problema$_3$: Impressão de números ímpares dentro do intervalo $[a, b]$.

IMPARES.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int a, b, i;
  clrscr();
  printf("Entre com valor inicial, a: ? ");
  scanf("%i", &a);
  printf("Enter com valor final, b: ? ");
  scanf("%i", &b);

  for (i=a; i<=b; i++){
    if (i%2!=0){
      // Note: se o resto da divisão não for nulo é porque o número
      //       é ímpar
      printf(" %i,", i);
    }
  }
  printf("\b \nFim\n");
}
```

Solução problema$_4$: Imprimir os N primeiros números da série de Fibonacci.

???.CPP

```
falta....
```

Solução problema$_5$: Joguinho da adivinhação de número sorteado pelo computador.

???.CPP
```
falta....
```

Parte III

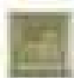
Matrizes & Strings

Curso de Engenharia Elétrica
Informática Aplicada à Engenharia Elétrica I

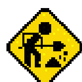
3ª Parte da Apostila de ANSI C

Matrizes & Strings

Prof. Fernando Passold



Museum of Fine Arts, Boston



Observação: Esta apostila está em fase de edição
Usando sistema de edição MiKTeX/LaTeX 2_ε para a mesma:
Ver: http://www.miktex.org/
Prof. Fernando Passold – Semestre 2005/1
Última atualização: 13 de março de 2006.



Capítulo 5

Vetores, Matrizes

Contents

Neste capítulo você aprenderá a lidar com vetores e matrizes de dados na linguagem C.

5.1 Conceito de *array*

Um "*array*" é uma coleção de dados do mesmo tipo referenciados usando um único nome de variável, na forma de um vetor ou matriz, tal qual se faz na matemática tradicional. Cada elemento deste vetor é acessado por meio um "índice", que tanto pode ser um valor constante (e fixo) quanto variável (uma variável inteira neste caso). Valores baixos de "índice" correspondem aos primeiros termos deste vetor, o valor mais alto para "índice" corresponde ao último elemento do vetor.

Por exemplo, na matemática tradicional, podemos criar o vetor A composto por apenas 8 elementos, como indicado abaixo:

$$A = [\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8\ ]$$

onde o primeiro elemento seria referenciado como:

$$A_1 = 1$$

o sétimo elemento seria acessado como:

$$A_7 = 7$$

note que os termos 1 e 7 usados correspondem aos "índices" que permitem acessar os elementos de um vetor.

5.2 Declaração de Arrays unidimensionais

Em C uma variável pode ser declarada como um array unidimensional fazendo-se:

```
tipo nome_var[tamanho];
```

Onde:

tipo: se refere ao tipo de dado que será manipulado por este array, podendo ser: *int*, *float*, *char* e suas variações (*long int*, *double*, etc);

nome_var: o nome que será adotado para esta variável que no caso é um array;

tamanho: define a quantidade de dados que este array irá guardar. Note que tamanho sempre será um valor inteiro.

Por exemplo, a matriz A usada como exemplo na seção anterior, poderia ser declarada como:

```
#include <stdio.h>
void main(){
  int A[8];
}
```

para o caso de se desejar trabalhar apenas com valores inteiros dentro da matriz A (no caso, a variável A).

Se fosse desejado armazenar valores $\epsilon \mathbb{R}$ (ponto flutuante), deveria ser feito:

```
void main(){
  float A[8];
}
```

A forma de acessar os elementos de um vetor em C é realizado da seguinte maneira:

Matemática Tradicional	Linguagem C	Observações
A_1	A[0]	primeiro elemento do array
A_7	A[6]	penúltimo elemento do array
A_8	A[7]	último elemento do array

Importante:



Note que em C, **todo array inicia na posição 0** [índice inicial = 0 (zero)] (e não à partir da posição 1 como ocorre em outras linguagens de programação).

Para atribuir valores à determinadas posições de um array, faz-se:

```
A[0]=1; A[6]=7; A[7]=8;
```

Note que neste caso, os índices utilizados para acessar os termos da matriz foram valores inteiros constantes, mas poderia ter sido utilizada uma variável como índice para acessar cada posição do array.

De fato, para inicializar a matriz A declarada anteriormente como:

$$A = [\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8\ ]$$

poderia ser feito em C, algo como:

```
#include <stdio.h>
void main(){
  int A[8], i;

  for (i=0; i<8; i++){
    A[i]= i+1;
  }
}
```

que é muito mais fácil que se atribuir valores à mão, termo à termo para cada um dos elementos da matriz (imagine o caso de um array contendo 50 elementos – você digitaria 50 linhas atribuindo valores para cada uma das 50 posições deste array?).

Pode-se imaginar os valores estocados num array, na forma de uma “caixinha” para cada um termos – de fato, cada elemento do array ocupa uma quantidade de bytes de acordo com o tipo de variável utilizado para definir este array – veja a figura 5.1 à seguir.

A[0]=	1
A[1]=	2
A[2]=	3
A[3]=	4
A[4]=	5
A[5]=	6
A[6]=	7
A[7]=	8

Figura 5.1: Ocupação de memória para uma variável `A` do tipo array.

Exibir os termos que compõem este array é simples, basta fazer:

Programa:

```
#include <stdio.h>
void main(){
  int A[8], i;

  for (i=0; i<8; i++){
    A[i]= i+1;
    printf("A[%i]= %i\n", i, A[i]);
  }
}
```

O que gera uma saída como:

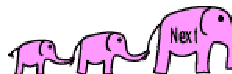
```
A[0]= 1
A[1]= 2
A[2]= 3
A[3]= 4
A[4]= 5
A[5]= 6
A[6]= 7
A[7]= 8
```

5.2.1 Inicialização de arrays

Em C, podemos inicializar um array já no momento da sua declaração.

Por exemplo, o caso anterior, da matriz A poderia ser implementado como:

```
int A[8] = { 1, 2, 3, 4, 5, 6, 7, 8};
```



5.2.2 Problemas

Problema$_1$: Crie um programa que inicialize a matriz B conforme indicado abaixo:

$$B = [\ 2 \quad 4 \quad 6 \quad 8 \quad 10 \quad 12\ ]$$

Problema$_2$: Crie um programa para armazenar os seguintes valores na matriz Y:

$$Y = [\ -1.5 \quad -1 \quad -0.5 \quad 0 \quad 0.5 \quad 1 \quad 1.5 \quad 2 \quad 2.5 \quad 3\ ]$$

Problema$_3$: Monte um programa que leia um conjunto X com N elementos reais e que calcule a diferença entre o maior e o menor elemento existente, além de indicar as posições em que eles ocorrem.

Problema$_4$: Monte um programa onde o usuário entra com o valor das diversas notas alcançadas por uma turma de alunos. O programa inicia perguntando o tamanho da turma e parte para a entrada de dados. A seguir, o programa deve ser capaz de exibir um histograma na tela, além de outras informações, conforme é mostrado à seguir:

```
Resultado da avaliação da TURMA A
    Menor nota:  4.0
    Maior nota: 10.0
Média da turma: 6.7

Histograma das notas:
0.0~ 3.0: ***
3.1~ 4.0: **
4.1~ 5.0: ****
5.1~ 6.0: *******
6.1~ 7.0: **********
7.1~ 8.0: *****
8.1~ 9.0: *
9.1~10.0: **
```

Note que 5 $\star$'s na linha "`7.1~ 8.0`" significa que 5 alunos alcançaram uma nota > 7.0 mas ≤ 8.0.

Problema$_5$: Monte um programa que realize o seguinte efeito visual conforme indicado abaixo:

Interação	Saída na tela
1	ooo
2	ooo
3	ooo
4	ooo
⋮	ooo
20	ooo

Note que este "efeito visual" é o que ocorre nos painéis de Led's usualmente empregados para exibir mensagens rolantes numa tela. As mensagens "correm" sempre da esquerda para a direita. O que ocorre na realidade é o deslocamento dos bits que correspondem aos Leds acessos no painel. O que se pede é que seja criado um programa que mostre na tela uma tabela exatamente igual à mostrada anteriormente, imaginando que temos a disposição um painel formado por 20 colunas de leds no sentido horizontal (existem outras tantas linhas de leds que completam o painel, mas neste caso, estamos interessados apenas numa das linhas).



5.2.3 Soluções dos Problemas

Problema$_1$) Solução:

ARRAY_P1.CPP

```
#include <stdio.h>
void main(){
  int B[6], i;

  for (i=0; i<6; i++){
    B[i]= 2+2*i;
    printf("B[%i]= %i\n", i, B[i]);
  }
}
```

Saída do programa:

```
B[0]= 2
B[1]= 4
B[2]= 6
B[3]= 8
B[4]= 10
B[5]= 12
```

Problema$_2$) Solução:

ARRAY_P2.CPP

```
#include <stdio.h>
void main(){
  int i;
  float Y[10];

  for (i=0; i<10; i++){
    //printf("A[%i]= %i\t->\t", i, A[i]);
    Y[i]= -1.5+0.5*i;
    printf("Y[%i]= %3.1f\n", i, Y[i]);
  }
}
```

Saída do programa:

```
Y[0]= -1.5
Y[1]= -1.0
Y[2]= -0.5
Y[3]=  0.0
Y[4]=  0.5
Y[5]=  1.0
Y[6]=  1.5
Y[7]=  2.0
Y[8]=  2.5
Y[9]=  3.0
```

Problema$_3$) Solução:

ARRAY_P2.CPP

```
#include <stdio.h>
void main(){
  int i;
  float Y[10];

  for (i=0; i<10; i++){
    //printf("A[%i]= %i\t->\t", i, A[i]);
    Y[i]= -1.5+0.5*i;
    printf("Y[%i]= %3.1f\n", i, Y[i]);
  }
}
```

Saída do programa:

```
Y[0]= -1.5
Y[1]= -1.0
Y[2]= -0.5
Y[3]=  0.0
Y[4]=  0.5
Y[5]=  1.0
Y[6]=  1.5
Y[7]=  2.0
Y[8]=  2.5
Y[9]=  3.0
```

Problema$_4$) Solução:

ARRAY_P3.CPP

```
#include<stdio.h>
void main(){
  float X[50], maior, menor, dif;
  int N, i, pos_maior=0, pos_menor=0;

  printf("Programa para ...\n\n");
  printf("Entre com N: ? ");
  scanf("%i", &N);
  if (N>50){
     printf("Desculpe, este programa lida no maximo com 50 valores\n\n");
  }
  else{
     printf("Entre com os %i valores:\n\n", N);
     for(i=0; i<N; i++){
       printf("Entre com o valor %i) ? ", i+1);
       scanf("%f", &X[i] );
       if (i==0){
          maior=X[0]; menor=X[0];
       }
       else{
    if (X[i]>maior){
       maior= X[i];
```

```
                pos_maior= i;
        }
        if (X[i]<menor){
           menor=X[i];
                pos_menor= i;
              }
           }
         } // fecha o for...
         printf("\n\nMaior valor: %5.2f - ocorre na posicao: %i\n", maior, pos_maior);
         printf(    "Menor valor: %5.2f - ocorre na posicao: %i\n", menor, pos_menor);
         dif=maior-menor;
         printf(    "Diferenca  : %5.2f\n", dif);
   }

}
```

Saída do programa:

```
Programa para ...

Entre com N: ? 5 Entre com os 5 valores:

Entre com o valor 1) ? 10
Entre com o valor 2) ? 7
Entre com o valor 3) ? 7.5
Entre com o valor 4) ? 5
Entre com o valor 5) ? 6


Maior valor: 10.00 - ocorre na posicao: 0
Menor valor:  5.00 - ocorre na posicao: 3
Diferenca  :  5.00
```

Problema$_5$) Solução:

ARRAY_P4.CPP

```
#include<stdio.h>
void main(){
  int alunos, i, maior, menor, j, h[10];
  float nota[50], soma, media;

  printf(".:Programa Estatistica de notas:...\n\n");
  printf("Entre com numero de alunos: ? ");
  scanf("%i", &alunos);
  if (alunos>50){
     printf("Desculpe, este programa lida no maximo com 50 alunos\n\n");
  }
  else{
     printf("Entre com as notas de cada aluno:\n\n");
     // inicializando variaveis...
     maior=0; menor=0; soma=0;
     for (i=0; i<10; i++){
       h[i]=0;
     }

     for(i=0; i<alunos; i++){
       printf("Nota do aluno %i) ? ", i+1);
       scanf("%f", &nota[i]);
       if (nota[i]>nota[maior]){
          maior=i;
       }
       if (nota[i]<nota[menor]){
          menor=i;
       }
       soma= soma+nota[i];

       // acumulando dados para histograma...
       for (j=0; j<10; j++){
         if (( nota[i]>j )&&( nota[i] <= (j+1) )){
            h[j]= h[j]+1;
            printf("Nota atual na faixa de %i ~ %i --> %i notas\n", j,j+1, h[j]);
         }
       }
       // Note: o if acima nao acumula notas = 0...
       if (nota[i]==0){
          h[0]= h[0]+1;
```

```
          printf("Nota atual na faixa de %i ~ %i --> %i notas\n", 0,1, h[0]);
        }

      } // fecha o for...

      media=soma/alunos;
      printf("\n\nResultados...\n\n");
      printf("Maior nota:  %5.2f (Aluno %i)\n", nota[maior], maior+1);
      printf("Menor nota:  %5.2f (Aluno %i)\n", nota[menor], menor+1);
      printf("Média turma: %5.2f\n\n", media);

      printf("Distribuicao das Notas\n\n");
      for (i=0; i<10; i++)
      {
        printf("%i.1 ~ %2i.0: ", i, i+1);
        for (j=0; j<h[i]; j++)
        {
           printf("*");
        }
        printf("\n");
      }
   }
   printf("\nFim do Programa\n");
}
```

Saída do programa:

```
.:Programa Estatistica de notas:...

Entre com numero de alunos: ? 7

Entre com as notas de cada aluno:

Nota do aluno 1) ? 5
Nota atual na faixa de 4 ~ 5 --> 1 notas
Nota do aluno 2) ? 5.5
Nota atual na faixa de 5 ~ 6 --> 1 notas
Nota do aluno 3) ? 6
Nota atual na faixa de 5 ~ 6 --> 2 notas
Nota do aluno 4) ? 10
Nota atual na faixa de 9 ~ 10 --> 1 notas
Nota do aluno 5) ? 8
Nota atual na faixa de 7 ~ 8 --> 1 notas
Nota do aluno 6) ? 9
Nota atual na faixa de 8 ~ 9 --> 1 notas
Nota do aluno 7) ? 8.5


Resultados...

Maior nota:  10.00 (Aluno 4)
Menor nota:   5.00 (Aluno 1)
Média turma:  7.43

Distribuição das Notas

0.1 ~  1.0:
1.1 ~  2.0:
2.1 ~  3.0:
3.1 ~  4.0:
4.1 ~  5.0: *
5.1 ~  6.0: **
6.1 ~  7.0:
7.1 ~  8.0: *
8.1 ~  9.0: **
9.1 ~ 10.0: *

Fim do Programa
```

5.3 Arrays Bidimensionais

Recordando conceitos de álgebra matricial, sejam as matrizes a e b:

$$a = \begin{bmatrix} 2 & 3 & 7 \\ 1 & -1 & 5 \end{bmatrix}$$

$$b = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 1 & 4 \\ 4 & 7 & 6 \end{bmatrix}$$

onde a é uma matriz de 2 linhas $\times$ 3 colunas e b é uma matriz 3×3 (3 linhas $\times$ 3 colunas). Cada um dos termos destas matrizes pode ser acessado da seguinte forma:

$$a = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$b = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

assim, o termo a_{23} corresponde ao elemento da 2ª linha e da 3ª coluna, ou ao valor 5. Genericamente então $b_{ij} = 4$ se $i = 3$ (3ª linha) e $j = 1$ (1ª linha), ou, $b_{31} = 4$. Note que estas matrizes são bidimensionais, ou seja, de dimensão 2, $\in \mathbb{R}^2$, ou também ditas, matrizes 2D [1].

Na linguagem C uma matriz pode ser declarada como se segue:

Sintaxe:

especificador_de_tipo *nome_da_matriz* [*tamanho_1*]...[*tamanho_N*];

onde:

especificador_de_tipo:	corresponde ao tipo de dados que serão guardados pela matriz, podendo ser: *int*, *float*, *double*, *char*, etc.;
nome_da_matriz	corresponde ao nome adotado para referenciar a variável do tipo matriz;
tamanho_i	especifica as dimensões da matriz;

Assim, por exemplo, as declarações das matrizes a e b, em C, ficariam como:

```
#include <stdio.h>
void main(){
  int a[2][3];
  float b[3][3];
}
```

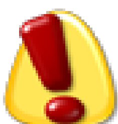
E para acessar cada um dos elementos destas matrizes, teríamos de fazer:

a[i][j]=

		j → 0	1	2
i ↓	0	2	3	7
	1	1	-1	5

ou:

$$\texttt{a[i][j]} = \begin{bmatrix} \texttt{a[0][0]} & \texttt{a[0][1]} & \texttt{a[0][2]} \\ \texttt{a[1][0]} & \texttt{a[1][1]} & \texttt{a[1][2]} \end{bmatrix}$$



Lembre-se: em C, os índices para acessar os elementos de uma matriz iniciam em 0 e não em 1!

[1]Um exemplo de especificação de matriz 3D, poderia ser: $c = 3 \times 3 \times 3$. Note que neste último caso, a matriz c estaria guardando $3 \times 3 \times 3 = 27$ elementos, enquanto as matrizes a e b estão estocando respectivamente: 6 e 9 elementos. Conforme a dimensão da matriz aumenta, perceba que aumenta exponencialmente o número de termos contidos dentro da mesma – isto implica em maior atenção no momento de se declarar uma matriz quando se está programando um computador.

Em C, este código ficaria como:

```
#include <stdio.h>
void main(){
  int a[2][3];
  float b[3][3];
  int i, j;

  a[0][0]=2;  a[0][1]=3;  a[0][2]=7;
  a[1][0]=1;  a[1][1]=-1; a[1][2]=5;

  b[0][0]=1;  b[0][1]=3;  b[0][2]=1;
  b[1][0]=2;  b[1][1]=1;  b[1][2]=4;
  b[2][0]=4;  b[2][1]=7;  b[2][2]=6;

  printf("Matriz a:\n\n");
  for (i=0; i<2; i++){
    for (j=0; j<3; j++){
      printf("%2i\t", a[i][j]);
    }
    printf("\n");
  }
  printf("\n");

  printf("Matriz b:\n\n");
  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      printf("%2.0f\t", b[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
```

Resultando em:

```
Matriz a:

 2        3        7
 1       -1        5

Matriz b:

 1        3        1
 2        1        4
 4        7        6
```

5.3.1 Inicialização alternativa de matrizes

Em C, outra forma de inicializar as matrizes a e b usadas anteriormente seria programar:

```
#include <stdio.h> void main(){

  int a[2][3] = {
    2,  3, 7,
    1, -1, 5
  };

  float b[3][3] = {
    1, 3, 1,
    2, 1, 4,
    4, 7, 6
  };

  int i, j;

  printf("Matriz a:\n\n");
  for (i=0; i<2; i++){
    for (j=0; j<3; j++){
      printf("%2i\t", a[i][j]);
    }
    printf("\n");
  }
  printf("\n");

  printf("Matriz b:\n\n");
  for (i=0; i<3; i++){
```

```
    for (j=0; j<3; j++){
      printf("%2.0f\t", b[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
```

Resulta também em:

```
Matriz a:

 2       3       7
 1      -1       5

Matriz b:

 1       3       1
 2       1       4
 4       7       6
```

5.3.2 Observações

Note que as matrizes mais utilizadas em computação são as matrizes bidimensionais. Mesmo para gerar gráficos 3D.

Por exemplo, para exibir um gráfico "mês $\times$ inflação", são necessárias apenas 2 dimensões. Uma matriz de 2 colunas $\times$ n linhas seria capaz de guardar os dados necessários para exibir este tipo de gráfico. A primeira coluna poderia estar relacionada com o mês (número do mês) e a segunda coluna, com o valor em porcentagem correspondendo à inflação daquele mês. Por exemplo, veja a tabela 5.1 e a figura 5.1 relacionada com o gráfico 2D resultante desta tabela de dados.

mês	%
janeiro	0,65
fevereiro	0,19
março	0,12
abril	0,29
maio	0,57
junho	0,92
julho	0,59
agosto	0,99
setembro	0,21
outubro	0,62
novembro	0,56
dezembro	0,67
janeiro	0,56
fevereiro	0,36
março	0,79
abril	0,83
maio	0,35

Tabela 5.1: Dados usados para gerar gráfico 2D – Inflação (IPC-FIPE).

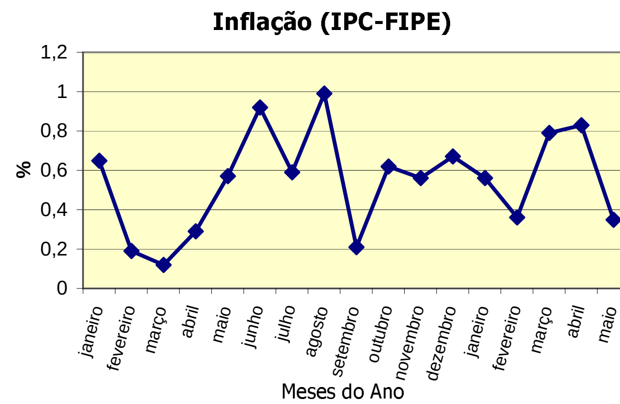


Figura 5.2: Exemplo de gráfico 2D.

Já um gráfico 3D como o mostrado na figura 5.3 à seguir exige apenas 2 colunas, ou seja, uma matriz 2D, contendo n linhas $\times$ 2 colunas, onde as linhas se referem às posições no eixo x e as colunas se referem às posições no eixo y. O elemento apontado pela linha x, coluna y se refere à altura z que a superfície deve possuir neste ponto (x, y) de intersecção no plano XY – ver tabela 5.2 à seguir.

		colunas $(x) \rightarrow$					
$z =$		10	11	12	13	14	15
	17	1.5069	2.7942	3.5855	3.6886	3.3404	2.9344
linhas	18	3.2876	4.9199	6.0650	6.3901	5.9370	5.0190
$\downarrow$	19	4.3742	6.1956	7.5413	7.9966	7.4805	6.2513
	20	4.2896	5.9599	7.2199	7.6723	7.1939	5.9862

Tabela 5.2: Parte da matriz z usada para gerar gráfico 3D.

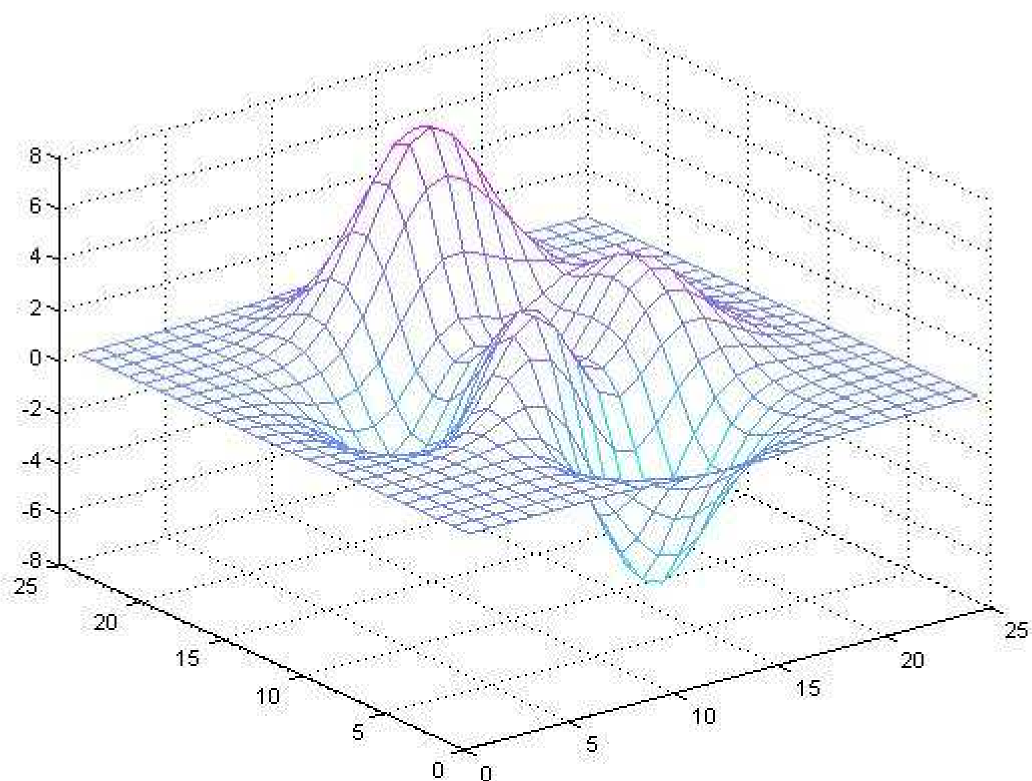


Figura 5.3: Exemplo de gráfico 3D.

A superfície mostrada na figura 5.3 foi obtida através da equação:

$$z = 3(1-x)^2 exp[-(x^2) - (y+1)^2] - 10(x/5 - x^3 - y^5)exp(-x^2 - y^2) - \frac{1}{3}exp[-(x+1)^2 - y^2]$$

5.3.3 Problemas

Dadas duas matrizes, a e b, conforme indicado abaixo:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad b = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

resolva o que se pede nos itens à seguir:

Problema$_1$: Monte um algoritmo capaz de inicializar a matriz a, usando apenas laços for.

Problema$_2$: Inicialize a matriz b usando apenas laços for.

Código para problemas 1 e 2:

MATRIZAB.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int a[2][3], b[3][3];
  int i, j, cont;

  // as linhas abaixo inicializam a[][]...
  clrscr();
  printf("matriz a:\n\n");
  cont=0;
  for(i=0; i<2; i++){
    for(j=0; j<3; j++){
      cont++;
      a[i][j]=cont;
      printf("%i\t", a[i][j]);
    }
    printf("\n");
  }
  // linhas abaixo, inicializam b[][]...
  cont=0;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++){
      cont++;
      b[j][i]=cont;
    }
  }
  // Note: mostro matriz b depois...
  printf("\nMatriz b:\n\n");
  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      printf("%2i\t", b[i][j]);
    }
    printf("\n");
  }
  printf("\nFim\n");

}
```

Saída do código:

```
Matriz a:

1        2        3 4        5        6

Matriz b:

 1        4        7
 2        5        8
 3        6        9

Fim
```

Problema$_3$: Monte um programa para inicializar as matrizes c e d conforme indicado abaixo:

$$c = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} \qquad d = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

Código:

MATRIZCD.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int c[3][3], d[3][3];
  int i, j, cont;

  clrscr();
  printf("Matriz c:\n\n");
  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
       c[i][j]=i+1;
       printf("%i\t", c[i][j]);
    }
    printf("\n");
  }

  printf("\nMatriz d:\n\n");
  for(i=0; i<3; i++){
    for(j=0; j<3; j++){
      d[i][j]= i+j+1;
      printf("%i\t", d[i][j]);
    }
    printf("\n");
  }

  printf("\nOk");
}
```

Saída do programa:

```
Matriz c:

1       1       1
2       2       2
3       3       3

Matriz d:

1       2       3
2       3       4
3       4       5

Ok
```

Problema$_4$: Monte um algoritmo para calcular o **determinante da matriz** b. ← Difícil

Solução:

Suponha a seguinte matriz, $b =$

$$b = \begin{bmatrix} b_{[0][0]} & b_{[0][1]} & b_{[0][2]} \\ b_{[1][0]} & b_{[1][1]} & b_{[1][2]} \\ b_{[2][0]} & b_{[2][1]} & b_{[2][2]} \end{bmatrix}$$

seu determinante pode ser calculado como:

$$\begin{array}{ccc} b_{[0][0]} & b_{[0][1]} & b_{[0][2]} \\ b_{[1][0]} & b_{[1][1]} & b_{[1][2]} \\ b_{[2][0]} & b_{[2][1]} & b_{[2][2]} \\ b_{[0][0]} & b_{[0][1]} & b_{[0][2]} \\ b_{[1][0]} & b_{[1][1]} & b_{[1][2]} \end{array}$$

onde a diagonal principal é calculada como:

$$Diag_Princ = \left\{ \begin{array}{cccccc} b_{[0][0]} & \times & b_{[1][1]} & \times & b_{[2][2]} & + \\ b_{[1][0]} & \times & b_{[2][1]} & \times & b_{[0][2]} & + \\ b_{[2][0]} & \times & b_{[0][1]} & \times & b_{[1][2]} & \end{array} \right.$$

e a diagonal secundária como:

$$Diag_Sec = \left\{ \begin{array}{cccccc} b_{[2][0]} & \times & b_{[1][1]} & \times & b_{[0][2]} & + \\ b_{[0][0]} & \times & b_{[2][1]} & \times & b_{[1][2]} & + \\ b_{[1][0]} & \times & b_{[0][1]} & \times & b_{[2][2]} & \end{array} \right.$$

e assim: $\Delta = Diag_Princ - Diag_Sec$.

Mas como calcular os elementos da $Diag_Princ$ e $Diag_Sec$ de uma forma mais "automatizada"???

(a) Note que na expressão que forma a soma relativa aos termos da diagonal principal, nos termos relacionados com a multiplicação a coluna varia de 0 à 2 e a linha vai sendo incrementada entre um termo e outro (inicia de 0):

$$\begin{array}{cccc} b[0][0] & b[1][1] & b[2][2] & + \\ b[1][0] & b[2][1] & b[0][2] & + \\ b[2][0] & b[0][1] & b[1][2] & \end{array}$$

Do ponto de vista de programação usando linguagem C, isto resultaria num algoritmo conforme destacado abaixo:

```
diag_princ=0.0;
for (l=0; l<n; l++) // n=3 -- ordem da matriz
{
  mult=1.0; aux=l; // aux começa na primeira linha....
  for (c=0; c<n; c++)
  {
    mult= mult*b[aux][c];
    aux++; // as linhas vão sendo incrementadas....
    if (aux>=n)
    {
      aux=0;
    }
  }
  diag_princ=diag_princ+mult;
}
printf("\nDiag_Princ= %f\n\n", diag_princ);
```

Um "teste de mesa" do algoritmo acima revela que conforme os laços de for(l=...) e for(c=...) vão sendo executados, a seguinte operação seria desenvolvida:

```
diag_princ=0.0; l= 0;
  mult= 1.0;
  aux= 0;
  c= 0;
    mult= 1.0*b[0][0];
    aux= 1;
  c= 1;
    mult= b[0][0]*b[1][1];
    aux= 2;
  c= 2;
    mult= b[0][0]*b[1][1]*b[2][2];
    aux= 3;
    aux= 0; // entrou no if
  diag_princ= 0 + b[0][0]*b[1][1]*b[2][2];
l= 1;
  mult= 1.0;
  aux= 1;
  c= 0;
    mult= 1.0*b[1][0];
    aux= 2;
  c= 1;
    mult= b[1][0]*b[2][1];
    aux= 3;
    aux= 0; // entrou no if
  c= 2;
    mult= b[1][0]*b[2][1]*b[0][2];
    aux= 1;
  diag_princ= b[0][0]*b[1][1]*b[2][2] +
              b[1][0]*b[2][1]*b[0][2];
l= 2;
  mult= 1.0;
  aux= 2;
  c= 0;
    mult= 1.0*b[2][0];
    aux= 3;
    aux= 0; // entrou no if
  c= 1;
    mult= b[2][0]*b[0][1];
    aux= 2;
  c= 2;
    mult= b[2][0]*b[0][1]*b[1][2];
```

```
    aux= 3;
    aux= 0; // entrou no if
  diag_princ= b[0][0]*b[1][1]*b[2][2] +
              b[1][0]*b[2][1]*b[0][2] +
              b[2][0]*b[0][1]*b[1][2];
```

(b) No caso da expressão que forma o somatório dos termos da diagonal secundário, ocorre algo um pouco parecido. As colunas de cada um dos termos da multiplicação varia de 0 à 2. E as linhas de cada um destes termos, inicia de 0 e vai sendo decrementado conforme a coluna avança:

$$\begin{array}{llll} b[0][0] & b[2][1] & b[1][2] & + \\ b[1][0] & b[0][1] & b[2][2] & + \\ b[2][0] & b[1][1] & b[0][2] & \end{array}$$

O programa completo ficaria:

MATRIZP3.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  float b[3][3] = {
    1, 3, 1,
    2, 1, 4,
    4, 7, 6
  };
  int l, c, aux, n=3; // n= dimensao da matriz
  float diag_princ, mult, diag_sec, det;

  clrscr();
  printf("Este programa calcula o determinante da matriz abaixo:\n\n");
  for (l=0; l<n; l++){
    for (c=0; c<n; c++){
      printf("%2.0f\t", b[l][c]);
    }
    printf("\n");
  }
  printf("\n");

  diag_princ=0.0;
  printf("Diag_Princ= ");
  for (l=0; l<n; l++)
  {
    mult=1.0; aux=l; // aux começa na primeira linha....
    for (c=0; c<n; c++)
    {
      mult= mult*b[aux][c];
      // mostrando na tela o que acontece...
      printf("b[%i][%i](%3.1f)*", aux,c,b[aux][c]);
      aux++; // as linhas vão sendo incrementadas....
      if (aux>=n)
      {
         aux=0;
      }
    }
    diag_princ=diag_princ+mult;
    // mostrando na tela o que acontece...
    //                              Diag_Princ= XXXX
    printf("\b(=%3.1f) + (=%3.1f)\n            ", mult,diag_princ);
  }
  clreol();
  printf("\nDiag_Princ= %f\n\n", diag_princ);

  diag_sec=0.0;
  printf("Diag_Sec=   ");

  for (l=0; l<n; l++)
  {
    mult=1.0; aux=l;
    for (c=0; c<n; c++)
    {
      mult= mult*b[aux][c];
      // mostrando na tela o que acontece...
      printf("b[%i][%i](%3.1f)*", aux,c,b[aux][c]);
      aux--;
      if (aux<0)
      {
```

```
        aux=n-1;
      }
    }
    diag_sec=diag_sec+mult;
    // mostrando na tela o que acontece...
    //                              Diag_Princ= XXXX
    printf("\b(=%3.1f) + (=%3.1f)\n            ", mult,diag_sec);
  }
  clreol();
  printf("\nDiag_Sec=   %f\n", diag_sec);

  det=diag_princ-diag_sec;
  printf("\nDeterminante = %f\n", det);
}
```

Saída do programa:

```
Este programa calcula o determinante da matriz abaixo:

 1       3       1
 2       1       4
 4       7       6

Diag_Princ= b[0][0](1.0)*b[1][1](1.0)*b[2][2](6.0)(=6.0) + (=6.0)
            b[1][0](2.0)*b[2][1](7.0)*b[0][2](1.0)(=14.0) + (=20.0)
            b[2][0](4.0)*b[0][1](3.0)*b[1][2](4.0)(=48.0) + (=68.0)

Diag_Princ= 68.000000

Diag_Sec=   b[0][0](1.0)*b[2][1](7.0)*b[1][2](4.0)(=28.0) + (=28.0)
            b[1][0](2.0)*b[0][1](3.0)*b[2][2](6.0)(=36.0) + (=64.0)
            b[2][0](4.0)*b[1][1](1.0)*b[0][2](1.0)(=4.0) + (=68.0)

Diag_Sec=   68.000000

Determinante = 0.000000
```

Problema$_5$: Monte um programa para realizar **soma de matrizes**: $c = b + b$. Note que são somas de matrizes.
Código:

MATRIZP3.CPP

```
#include <stdio.h>
void main(){
  int b[3][3], c[3][3];
  int i, j, cont;

  // linhas abaixo, inicializam b[][]...
  cont=0;
  for(i=0; i<3; i++)
  {
    for(j=0; j<3; j++)
    {
      cont++;
      b[j][i]=cont;
    }
  }

  // adição....
  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      c[i][j]= b[i][j] + b[i][j];
    }
  }

  // linhas abaixo: imprimem conteudo da matriz c
  printf("Matriz c:\n\n");
  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      printf("%2i\t", c[i][j]);
    }
    printf("\n");
  }
  printf("\n");
```

```

}
```

Resultado:

```
Matriz c:

 2       8       14
 4      10       16
 6      12       18
```

Problema$_6$: Monte um programa que realize a **multiplicação**: $d = a \times b$. Repare que o resultado deve ser uma matriz d de 2×3.

Solução:
Suponha que as matrizes a e b sejam do tipo: $a_{n \times m}$ e $b_{m \times p}$.
Notamos que primeiramente, para que esta multiplicação possa ser realizada é necessário que o número de colunas da matriz a seja igual ao número de linhas da matriz b. O que garantimos na hipótese acima, através da variável m. A matriz c gerada pela multiplicação das matrizes a e b ao final vai ser de dimensão: $c_{n \times p}$.

Podemos dar início agora à lógica que há por trás da multiplicação de matrizes. Vamos continuar supondo nossas matrizes: $c_{n \times p} \leftarrow a_{n \times m} \times b_{m \times p}$.

Realizando alguns passos desta multiplicação “manualmente” obtemos:

$$\underbrace{\begin{bmatrix} c[0][0] & c[0][1] & c[0][2] \\ c[1][0] & c[1][1] & c[1][2] \end{bmatrix}}_{c_{n \times p}} = \underbrace{\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \end{bmatrix}}_{a_{n \times m}} \times \underbrace{\begin{bmatrix} b[0][0] & b[0][1] & b[0][2] \\ b[1][0] & b[1][1] & b[1][2] \\ b[2][0] & b[2][1] & b[2][2] \end{bmatrix}}_{b_{m \times p}}$$

Para obtermos o termo $c_{1,1}$, na notação da linguagem C, ficaria:

$$c_{1,1} = c[0][0] = a[0][0] * b[0][0] + a[0][1] * b[1][0] + a[0][2] * b[2][0]$$

outros termos:

$$c_{1,2} = c[0][1] = a[0][0] * b[0][1] + a[0][1] * b[1][1] + a[0][2] * b[2][1]$$

e:

$$c_{2,1} = c[1][0] = a[1][0] * b[0][0] + a[1][1] * b[1][0] + a[1][2] * b[2][0]$$

Note, pelas diferentes cores adotadas anteriormente (e de forma proposital), o que acontece com relação aos índices que acessam cada um dos termos das matrizes responsáveis pelas multiplicações anteriores

Note que a linha da matriz a varia conforme a linha que está sendo computada da matriz b (cor “■”). Da mesma forma, perceba que a coluna da matriz b varia conforme varia a coluna da matriz c sendo computada (cor “■”). Note ainda que existe um “laço mais interno”, variando um 3° índice que se refere aos termos sendo somados – note que enquanto varia a coluna da matriz a, varia da mesma forma a linha da matriz b (cor “■”).

Isto por fim resulta no fluxograma para cômputo da multiplicação de matrizes mostrada na figura 5.4.

Código:

MULT.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  int a[2][3]={ 1, 2, 3,
```

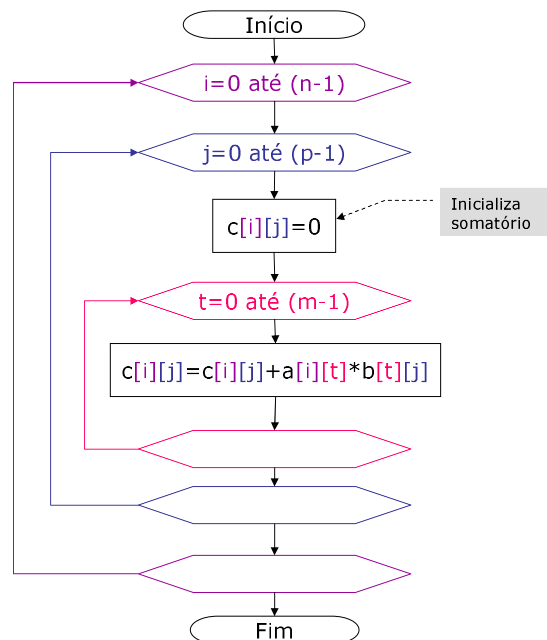


Figura 5.4: Fluxograma básico de multiplicação de matrizes.

```
                 4, 5, 6};
  int b[3][3]={ 1, 4, 7,
                2, 5, 8,
                3, 6, 9};
  int c[2][3];
  int i, j, t;

  // multiplicaþÒo abaixo:
  for (i=0; i<2; i++)
  {
    for (j=0; j<3; j++)
    {
      c[i][j]= 0;
      for (t=0; t<3; t++)
      {
        c[i][j]= c[i][j] + a[i][t]*b[t][j];
      }
    }
  }

  // imprimindo resultado:
  clrscr();
  printf("Matriz c resultante:\n\n");
  for (i=0; i<2; i++){
    for (j=0; j<3; j++){
      printf("%3i  ", c[i][j]);
    }
    printf("\n");
  }

  printf("\nFim");
}
```

Saída:

```
Matriz c resultante:

 14   32   50
 32   77  122

Fim
```

Problema$_7$: Monte um programa que realize **multiplicação de matrizes genéricas**. O programa pergunta ao usuário as dimensões de cada uma das matrizes, verifica se a multiplicação é possível. Em caso

afirmativo, o programa pede para o usuário informar o elementos de cada uma das matrizes e mostra o resultado da multiplicação das duas matrizes.
Código do programa:

MULT2.CPP

```
// Programa para multiplicar matrizes quadradas
#include <stdio.h>
#include <conio.h>
#define TAM_MAX 10
#define TRUE  1
#define FALSE 0
void main(){
  float a[TAM_MAX][TAM_MAX]; // matriz de entrada de dados
  float b[TAM_MAX][TAM_MAX]; // matriz de entrada de dados
  float c[TAM_MAX][TAM_MAX]; // matriz resultante
  int n; // linhas matriz A
  int m; // colunas matriz B
  int p; // colunas matriz B
  int i, j, t; // variaveis auxiliares para loopings internos
  int problemas=FALSE; // indica se houve problemas com entrada de dados

  // Segue entrada de dados
  // primeiro os dados das matrizes a serem multiplicadas
  clrscr();
  printf(":: Multiplicacao de Matrizes ::\n\n");
  printf("Obs: De dimensoes ate %i x %i\n\n", TAM_MAX, TAM_MAX);
  printf("Entre com numero de linhas  da matriz A: ");
  scanf("%i", &n);
  if (n>TAM_MAX){
    problemas=TRUE;
  }
  if (!problemas){
    printf("Enter com numero de colunas da matriz A: ");
    scanf("%i", &m);
    if (n>TAM_MAX){
      problemas=TRUE;
    }
  }
  if (!problemas){
    printf("\nEstou supondo que a matriz B contem %i linhas\n", m);
    printf("Entre agora com numero de colunas da matriz B: ? ");
    scanf("%i", &p);
    if (n>TAM_MAX){
      problemas=TRUE;
    }
  }
  if (problemas){
    printf("\nERRO: Sinto muito, este programa trabalha com no maximo\n");
    printf("matrizes de %i x %i\n", TAM_MAX, TAM_MAX);
    printf("\nPrograma abortado\n");
    printf("Tente entrar com matrizes menores\n");
  }
  else{
    printf("\nPois bem, procedendo a entrada de dados da matrizes\n\n");

    // entrada de dados da matriz A:
    for (i=0; i<n; i++){
      for (j=0; j<m; j++){
        printf("A(%i,%i)= ? ", i+1, j+1);
        scanf("%f", &a[i][j]);
      }
      printf("\n"); // pausa entre linhas da matriz...
    }

    // entrada de dados da matriz B:
    for (j=0; j<m; j++){
      for (t=0; t<p; t++){
        printf("B(%i,%i)= ? ", j+1, t+1);
        scanf("%f", &b[j][t]);
      }
      printf("\n"); // pausa entra linhas da mariz...
    }

    printf("\nMultiplicando matrizes");
    // multiplicação abaixo:
    for (i=0; i<2; i++){
      for (j=0; j<3; j++){
```

```
        c[i][j]= 0; // inicializando soma com zero
        for (t=0; t<3; t++){
          c[i][j]= c[i][j] + a[i][t]*b[t][j]; // multiplicando termos...
          printf("."); // apenas gera efeito visual - calculando...
        }
      }
    }

    // imprimindo resultado:
    printf("\n\nResultado: matriz C: %i x % i:\n\n", n, p);
    for (i=0; i<2; i++){
      for (j=0; j<3; j++){
        printf("%7.2f  ", c[i][j]);
      }
      printf("\n");
    }

    printf("\nFim");
  }
}
```

Exemplo de saída do programa:

```
:: Multiplicacao de Matrizes ::

Obs: De dimensoes ate 10 x 10

Entre com numero de linhas  da matriz A: 2 Enter com numero de
colunas da matriz A: 3

Estou supondo que a matriz B contem 3 linhas Entre agora com numero
de colunas da matriz B: ? 3

Pois bem, procedendo a entrada de dados da matrizes

A(1,1)= ? 1 A(1,2)= ? 2 A(1,3)= ? 3

A(2,1)= ? 4 A(2,2)= ? 5 A(2,3)= ? 6

B(1,1)= ? 1 B(1,2)= ? 4 B(1,3)= ? 7

B(2,1)= ? 2 B(2,2)= ? 5 B(2,3)= ? 8

B(3,1)= ? 3 B(3,2)= ? 6 B(3,3)= ? 9


Multiplicando matrizes.................

Resultado: matriz C: 2 x  3:

  14.00    32.00    50.00
  32.00    77.00   122.00

Fim
```

Note que: o programa anterior está preparado para o caso do usuário tentar multiplicar matrizes cujo tamanho superam a capacidade do programa. Neste caso, surge uma mensagem de erro como a mostrada no exemplo à seguir:

```
:: Multiplicacao de Matrizes ::

Obs: De dimensoes ate 10 x 10

Entre com numero de linhas  da matriz A: 20

ERRO: Sinto muito, este programa trabalha com no maximo matrizes de
10 x 10

Programa abortado Tente entrar com matrizes menores
```

Note ainda que: no código, a variável `problemas` faz o papel de uma variável booleana (só pode conter 2 valores: `TRUE` ou `FALSE` – Verdadeiro ou Falso), e que a entrada de dados à respeito das

dimensões das matrizes só continua enquanto o limite TAM_MAX não é superado – linhas 27 e 34. Note que isto pode acontecer tanto com o número de linhas ou de colunas de alguma das matrizes, por isto, o teste para verificar o tamanho deve ser realizado sempre depois da entrada de cada um destes dados (linhas 24–26, 30–32 e 38–40). **E mais**: a entrada dos dados restantes que faltam para completar as informações à respeito das dimensões das matrizes só continua caso o número de linhas ou de colunas para alguma das matrizes não tenha ultrapassado o valor máximo. Caso este valor tenha sido ultrapassado, note que a variável `problemas` passa a ser `TRUE` e repare então que o programa não vai mais continuar pedindo para o usuário entrar com os próximos dados à respeito das dimensões das matrizes.



5.4 Strings – um caso especial de array de char

O uso mais comum de matrizes unidimensionais (vetores), em C, é como string de caracteres. Em C, uma string é definida como sendo uma matriz (um array) de caracteres que é terminada com o valor nulo (zero). Este **caracter nulo** é especificado como '\0'. Por isto, é necessário se declarar as strings (arrays de caracteres) como sendo **1 caractere mais longo** que o comprimento da maior palavra ou frase que poderá estar contido dentro desta string. Isto porque a linguagem C não contempla o tipo de dado '*string*'. Mas permite constantes string que são listas de caracteres estre aspas, por exemplo: "alo aqui".

Exemplo: para declarar uma string capaz de guardar 10 caracteres teria de ser declarado algo como:

```
char str[11];
```

isto forçaria organizar a memória da seguinte forma:

	0	1	2	3	4	5	6	7	8	9	10
str =											\0

Detalhe: porém isto não significa que você (o programador) precisa adicionar à mão o caracter nulo, o compilador C faz isto por você automaticamente.

5.4.1 Inicialização de strings

Strings podem ser inicializandas no próprio instante da sua declaração, por exemplo:

```
char msg[19] = "Isto é uma string!";
```

Porém, não confundir **variáveis string** com **variáveis do tipo** ***char***, por exemplo, para inicializar uma variável do tipo *char*, você teria de fazer algo como:

```
char tecla = 'S';
```

Uma variável do tipo string poderia ainda ser inicializada como:

```
char msg[19] = { 'I', 's', 't', 'o', ' ', 'é', ' ', 'u', 'm', 'a', ' ', 's', 't', 'r', 'i' , 'n', 'g', '!', '\0' };
```

porém note que neste último caso, é você que deve inserir o caracter nulo ao final da string.

O exemplo anterior deixa claro a forma como a string "msg" vai ficar organizada na memória com computador:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
msg =	'I'	's'	't'	'o'	' '	'é'	' '	'u'	'm'	'a'	' '	's'	't'	'r'	'i'	'n'	'g'	'!'	'\0'

Isto quer dizer que cada elemento da string pode ser acessado de forma independente, por exemplo: `msg[8]='m'`.

Da mesma forma, perceba que é você (o programador) que deve ter certeza de declarar um array de char longo o suficiente para incluir a string desejada, caso contrário, seu programa corre o risco de perder-se durante a execução do mesmo (o tipo de erro conhecido como "***run-time error***" – note que estes tipos de erros são difíceis de se descobrir em grandes programas).

5.4.2 Inicialização de strings não-dimensionadas

Imagine que você queira inicializar certos tipos de mensagens padrão no seu programa, por exemplo, uma tabela de mensagens de erro, como mostrado abaixo:

```
char erro1[17]= "erro de leitura\n";
char erro2[17]= "erro de escrita\n";
char erro3[29]= "arquivo não pode ser aberto\n";
```

Note que é ligeiramente tedioso contar os caracteres presentes em cada mensagem. Você pode deixar o próprio Compilador C determinar automaticamente as dimensões de cada um dos arrays de *char* usando o conceito de vetores não dimensionados. Desta forma, o Compilador C se vê obrigado a criar vetores grandes o suficiente para conter todos os caracteres dos vetores sendo declarados. Usando esta abordagem, o exemplo anterior fica assim:

```
char erro1[]= "erro de leitura\n";
char erro2[]= "erro de escrita\n";
char erro3[]= "arquivo não pode ser aberto\n";
```

Este método além de ser menos tedioso, permite que o programador altere qualquer mensagem sem se preocupar em declarar um vetor com a dimensão correta.

5.4.3 Saídas de dados do tipo string

Note que para imprimir o conteúdo de variáveis do tipo string, necessitamos tomar alguns cuidados.

Se for usado o comando "printf()", o especificador de tipo de dado à ser exibido deve ser "\%s" (o 's' indica se tratar de uma string). A função printf vai exibir então todos os caracteres presentes dentro da matriz de caracteres sendo avaliada, até que o caracter nulo seja encontrado (este caracter evidentemente não é impresso; e tão pouco a função printf gera uma quebra de linha automaticamente - caracter '\n').

Existe outra função (comando) que pode ser utilizada para imprimir o conteúdo de uma string: a função "puts()". Por exemplo:

testeput.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  char msg[]= "Teste";
  clrscr();
  printf("Testando impressão de strings\n\n");
  printf("Teste com printf:\n");
  printf("%s\n\n", msg);
  printf("Teste com puts:\n");
  puts(msg);
  puts(" <- Note o que acontece aqui!");
}
```

Saída do programa anterior:

```
Testando impressão de strings

Teste com printf: Teste

Teste com puts: Teste
 <- Note o que acontece aqui!
```

Note que a função puts() imprime automaticamente o caracter '\n' no final da string, gerando a mudança automática de linha.

A função puts() é mais rápida que uma função printf() porque puts() está preparada para escrever apenas string de caracteres – não pode escrever números ou fazer conversões de formato como printf().

5.4.4 Entradas de dados do tipo string

A função scanf() é normalmente utilizada para passar dados lidos através do teclado para dentro da variável declarada como segundo argumento da função scanf() (após o primeiro argumento que identifica o tipo de dado à ser lido).

Já estudados como ler dados para uma variável do tipo *char*:

```
char tecla;

printf("Digite Sim/Não para continuar no programa: ");
scanf("%c", &tecla);
```

Note que para ler uma string usando o comando scanf() é necessário usar o especificador de formato "%s", justamente para "avisar" esta função de que deve realizar uma seqüencia de leitura de caracteres. Além disto, para o caso específico de leitura de dados para uma variável do tipo string, não deve ser usado o caracter "&" antes do segundo argumento da função scanf(). Porém a função scanf() lê caracteres digitados à partir do teclado até que seja encontrado um caractere de espaço em branco (além do ENTER ou RETURN que o usuário deve digitar para caracterizar o final da entrada de dados). Por exemplo:

testscan.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  char nome[60];
  clrscr();
  printf(":: :: Testando entrada de dados para string :: ::\n\n");
  printf("Tente entrar com seu nome completo: ? ");
  scanf("%s", nome);
  printf("\nConteudo da variavel nome: %s\n", nome);
}
```

Saída gerada:

```
:: :: Testando entrada de dados para string :: ::

Tente entrar com seu nome completo: ? Fernando Passold

Conteudo da variavel nome: Fernando
```



Como resolver este problema então ???

Nestes casos, o melhor é utilizar outra função do C para ler strings à partir do teclado, como por exemplo, a função gets().

A função gets() lê uma string de caracteres inserida pelo teclado e a devolve para a variável string passada como argumento da função. Por exemplo:

testgets.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  char nome[60];
  clrscr();
  printf(":: :: Testando entrada de dados para string :: ::\n\n");
  printf("Tente entrar com seu nome completo: ? ");
  // scanf("%s", &nome);
  gets(nome);
  printf("\nConteudo da variavel nome: %s\n", nome);
}
```

Saída gerada:

```
:: :: Testando entrada de dados para string :: ::

Tente entrar com seu nome completo: ? Fernando Passold

Conteudo da variavel nome: Fernando Passold
```

5.4.5 Operadores especiais com strings – biblioteca <string.h>

Outros detalhes da linguagem C: em C, não é possível se fazer atribuições diretas de valores para variáveis string. Por exemplo, tente fazer:

STRING1.CPP

```
#include <stdio.h>
#include <conio.h>
void main(){
  char msg[60];
  clrscr();
  printf(":: :: Testando atribuicao de dados para string :: ::\n\n");
  msg="Teste";
  printf("\nConteudo da variavel msg: %s\n", msg);
}
```

Deve surgir o seguinte erro (de compilação):

```
 Compiling STRING1.CPP:
*Error STRING1.CPP 7: Lvalue required
```



Como passar valores para variáveis do tipo string ???

RESPOSTA: para atribuir valores para uma variável do tipo string é necessário se apelar para a função strcpy() declarada dentro da biblioteca `<string.h>`.

O programa anterior fica então como:

STRING1.CPP

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(){
  char msg[60];
  clrscr();
  printf(":: :: Testando atribuicao de dados para string :: ::\n\n");
  // msg="Teste";
  strcpy(msg,"Teste");
  printf("\nConteudo da variavel msg: %s\n", msg);
}
```

A seguinte saída deve ser gerada agora:

```
:: :: Testando atribuicao de dados para string :: ::


Conteudo da variavel msg: Teste
```

Assim como no caso da função strcpy(), existem outras funções especialmente desenvolvidas para lidar com strings no C. Elas estão declaradas na biblioteca `<string.h>`. As principais funções desta biblioteca são mostradas na tabela à seguir:

Função	**Comentários**
strcpy(s1, s2)	Copia s2 em s1, ou, $s1 \leftarrow s2$
strcat(s1, s2)	Adiciona s2 ao final de s1, ou, $s1 = s1 + s2$
strlen(s1)	Retorna o tamanho de s1
strcmp(s1, s2)	Comparação de Strings. Retorna 0 se $s1 = s2$; Um número < 0 se $s1 < s2$; um número > 0 se $s1 > s2$

5.4.6 Outros exemplos

Ex.1) Aumentando o conteúdo de uma variável string:

STRING2.CPP

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(){
  char msg1[40], msg2[40];
  clrscr();
  printf(":: :: Testando composicao de strings :: ::\n\n");
  printf("Entre com um nome: ? ");
  gets(msg1);
  printf("Entre com um segundo nome: ? ");
  gets(msg2);
  strcat(msg1, msg2);
  printf("\nJuntando os 2 nomes: %s", msg1);
}
```

Saída do programa:

```
:: :: Testando composicao de strings :: ::

Entre com um nome: ? Joao Entre com um segundo nome: ? Silva

Juntando os 2 nomes: JoaoSilva
```

Note que: o conteúdo original de '`msg1`' foi "perdido" – esta string aumentou de tamanho.

5.4.7 Matrizes de strings

Da mesma forma que se declaram matrizes numéricas é possível se declarar matrizes de strings.

Para criar uma matriz de strings, declare uma matriz bidimensional de caracteres. Neste caso, o tamanho do índice do lado esquerdo da matriz indicaria o número de strings possíveis e o tamanho do índice do lado direito, especificaria a quantidade máxima de caracteres possíveis de serem estocadas em cada linha string desta matriz. Por exemplo:

```
char texto[30][80];
```

o código anterior declara uma variável de nome "`texto`" do tipo matriz de strings, capaz de guardar o equivalente à 30 linhas de texto com comprimento máximo de 79 caracteres cada linha (não se esqueça de reservar 1 espaço para o caracter nulo: '\0').

Para acessar cada linha desta matriz, você teria que especificar apenas o índice esquerdo. Por exemplo, o seguinte comando mostraria apenas a terceira linha da variável `texto` declarada anteriormente:

```
printf("%s", texto[2]);
```

Da mesma forma, passar dados do teclado para a quinta linha da matriz `texto` declarada anteriormente, poderia ser feito como:

```
gets(texto[4]);
```

5.5 Matrizes multidimensionais

C permite matrizes com mais de duas dimensões. O limite exato vai depender do compilador C utilizado. A forma genérica de declaração de uma matriz multidimensional é:

```
especificador_de_tipo nome_da_matriz [tamanho_1][tamanho_2][tamanho_3]...[tamanho_N];
```

Matrizes de 3 ou mais dimensões não são utilizadas com freqüência devido à quantidade de memória que elas necessitam. Por exemplo, uma matriz de quatro dimensões do tipo caractere e de tamanho: $10 \times 6 \times 9 \times 4$ requer $10 \times 6 \times 9 \times 4 \times 1$ byte = 2.160 bytes.

A figura 5.5 mostra um exemplo de matriz 3D, de dimensão: de $3 \times 3 \times 3$, onde alguns elementos foram ressaltados.

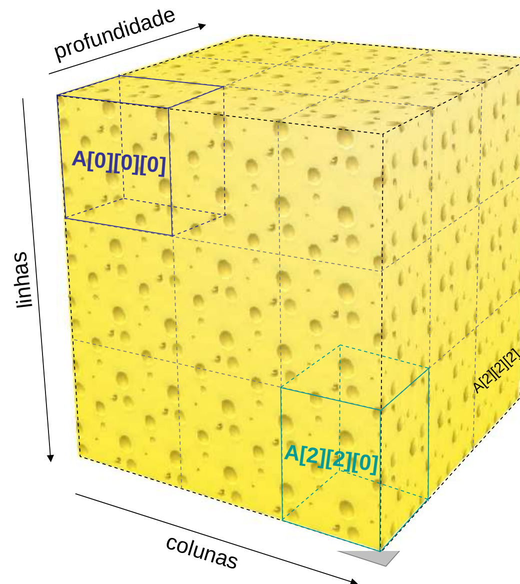


Figura 5.5: Exemplo de matriz 3D

Grandes matrizes multidimensionais são geralmente alocadas dinamicamente, uma parte por vez, com funções de alocação dinâmica de memória disponível na linguagem C e usando ponteiros. Esta abordagem é chamada de *matriz esparsa*. Neste tipo de matriz nem todos os elementos estão realmente presentes ou são necessários. Estas matrizes estão em grande parte "vazias". Esta técnica (matrizes esparsas) é considerada quando 2 condições acontecem: a) as dimensões da matriz são relativamente grandes (possivelmente acima da memória disponível no equipamento onde será rodada a aplicação) e b) nem todas as posições da matriz são utilizadas. Um exemplo típico de aplicação de matriz esparsa é um programa de planilha eletrônica. Embora virtualmente uma planilha eletrônica possa ter capacidade para lidar com 999 linhas por 999 colunas (por exemplo), isto não significa que realmente foi alocado fisicamente memória para conter $999 \times 999 = 998.001$ elementos. Existe então o que chamamos de *matriz lógica* e *matriz física* (ou real). A matriz física é que realmente existe dentro da memória do computador e corresponde à aproximadamente apenas às células realmente ocupadas na planilha eletrônica. A aplicação (programa da planilha eletrônica) "simula" para o usuário uma planilha real de 999 linhas × 999 colunas (seria a matriz lógica). Naturalmente existem técnicas de programação para realizar mapeamentos entre o que seria a matriz lógica (enxergada pelo usuário) e a matriz física (a realmente alocada dentro da memória do computador). Não faz parte do escopo desta disciplina estudar estas técnicas agora. Entre elas: lista encadeada, árvore binária, matriz de ponteiros e fragmentação.

5.6 Problemas Finais

P1) Monte um algoritmo para inicializar matrizes como indicado abaixo:

a) Matriz a:

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

b) Matriz b:

$$b = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 & 20 & 21 \end{bmatrix}$$

c) Matriz c:

$$c = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 14 & 13 & 12 & 11 & 10 & 9 & 8 \\ 15 & 16 & 17 & 18 & 19 & 20 & 21 \end{bmatrix}$$

d) Matriz d:

$$d = \begin{bmatrix} 1 & 8 & 9 & 16 & 17 & 24 \\ 2 & 7 & 10 & 15 & 18 & 23 \\ 3 & 6 & 11 & 14 & 19 & 22 \\ 4 & 5 & 12 & 13 & 20 & 21 \end{bmatrix}$$

e) Matriz e:

$$e = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

P2) E se agora fosse pedido para gerar uma matriz f, como a mostrada à seguir, onde o usuário entra com a posição (linha e coluna) do seu "elemento central" (contêm valor 1)?

$$f = \begin{bmatrix} 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 5 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 4 & 3 & 3 & 3 & 3 & 3 \\ 5 & 4 & 3 & 2 & 2 & 2 & 3 \\ 5 & 4 & 3 & 2 & 1 & 2 & 3 \\ 5 & 4 & 3 & 2 & 2 & 2 & 3 \\ 5 & 4 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Note que no caso anterior, a posição do elemento central é: $f_{5,5} = 1$ ($= f[4][4]$ na notação da linguagem C). [Dificuldade: Difícil]

P3) Monte um programa que leia 10 números inteiros fornecidos pelo usuário e que depois, separe numa outra variável (um vetor), apenas os números que forem divisíveis por 3.

P4) Faça um programa que gere aleatóriamente n números inteiros positivos variando entre 1 e 10 e que depois, separe estes números em 3 vetores diferentes. No primeiro vetor, A, devem ser colocados o números menores que 3 (excluindo o 3, $x < 3$); no vetor B, devem ser colocados os números entre 3 e 7 (ou, $3 \leq x \leq 7$), e no vetor C, o números maiores e iguais à 7 ($x > 7$). [Dificuldade média]

P5) Monte um programa capaz de preencher uma matriz de 10×10, colocando 0 (zero) nas posições onde $linha + coluna$ formam um número par e 1 (um) nas outras posições.

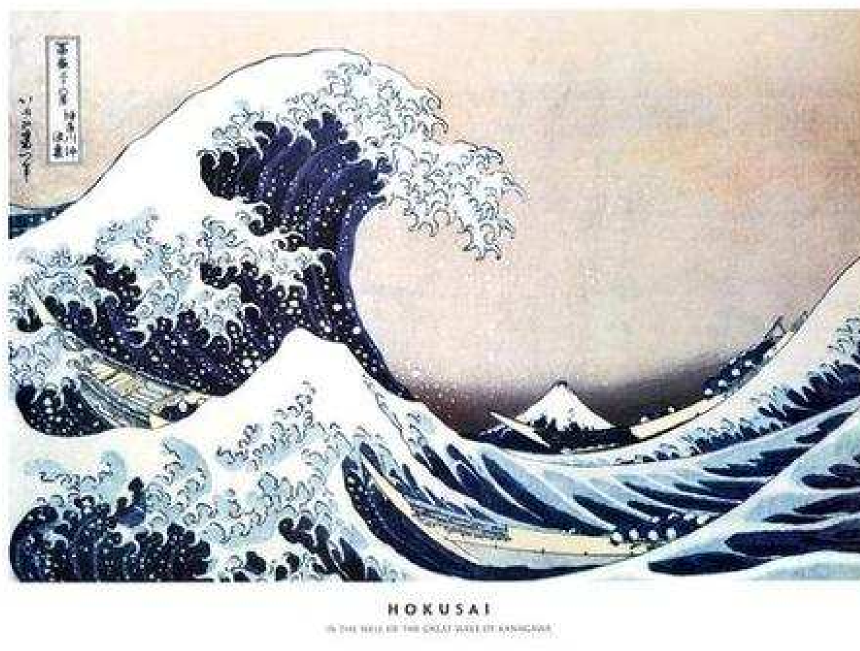
P6) Monte um programa para determninar a transposta de uma matriz quadrada.

P7) Monte um programa que pede ao usuário o tamanho desejado para a matriz; à seguir, o programa pede o número da linha inicial à partir da qual será gerado um efeito visual como o mostrado abaixo: por exemplo, neste caso, a matriz é de 10×7 e a linha inicial = 5.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

P8) Monte um programa que crie uma variação visual em relação ao problema anterior. Neste caso, é desejado um efeito visual como o que segue:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$



Apêndice A

Ferramentas de Eng. de Software

Na internet encontram-se duas ótimas ferramentas para projeto de sistemas em UML. A primeira delas, a ArgoUML, é uma ferramenta livre e tem o código fonte disponível, entretanto é um pouco instável e temperamental. O endereço pra quem se interessar é: "`http://argouml.tigris.org`" (Disponível em 07/03/2005). Esta ferramenta é bem completa e gera inclusive código em Java para os projetos nela desenvolvidos, realiza engenharia reversa e utiliza o padrão XSI como formato para armazenamento dos diagramas. A figura A.1 mostra uma das janelas deste *software*.

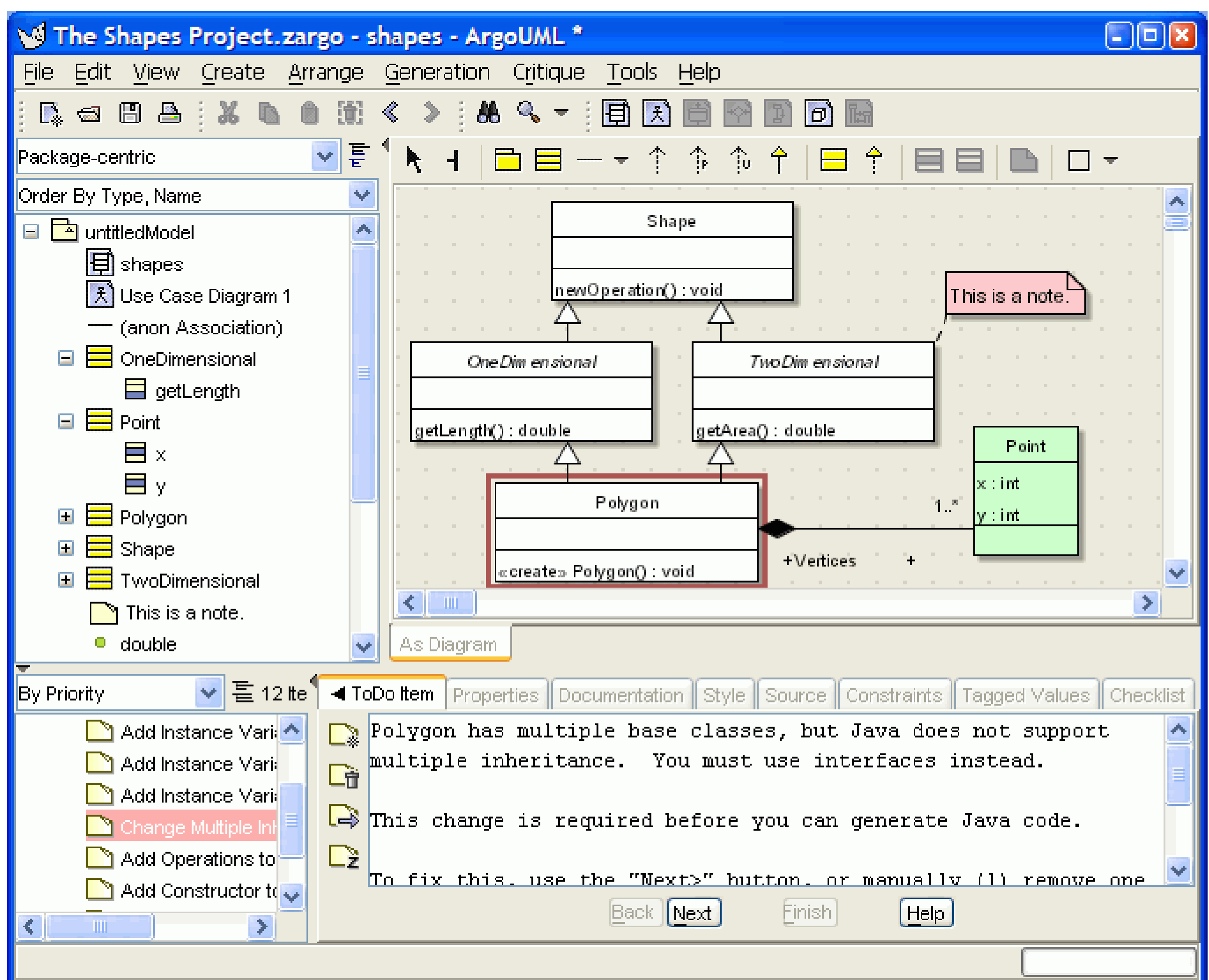


Figura A.1: Tela do *software* ArgoUML.

A outra ferramenta: **poseidon** for uml, é uma extensão comercial da ArgoUML. Entretanto a versão *Community Edition 3.0* pode ser baixada livremente (*free*) e não tem qualquer tipo de limi-

tação. Todos os diagramas e elementos da UML estão lá. O endereço pra quem se interessar é: "`http://www.gentleware.com/products/download.php3`" (Disponível em 07/03/2005). Uma grande vantagem da poseidon para a ArgoUML é que esta implementa o padrão UML de maneira mais rígida e é mais estável. A figura A.2 mostra algumas telas do *software* poseidon.

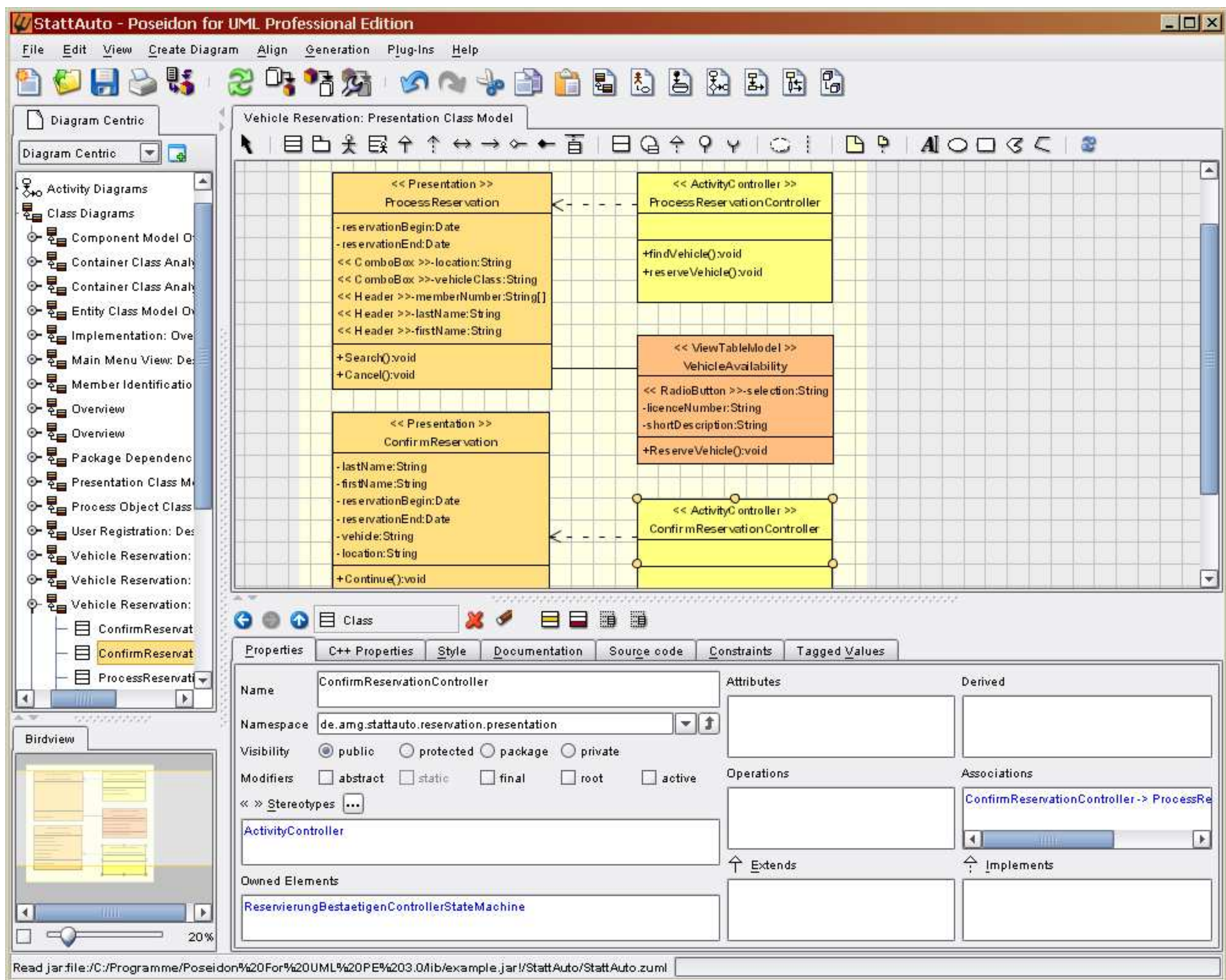


Figura A.2: Posseidon: Diagrama de Classes – Propriedades.

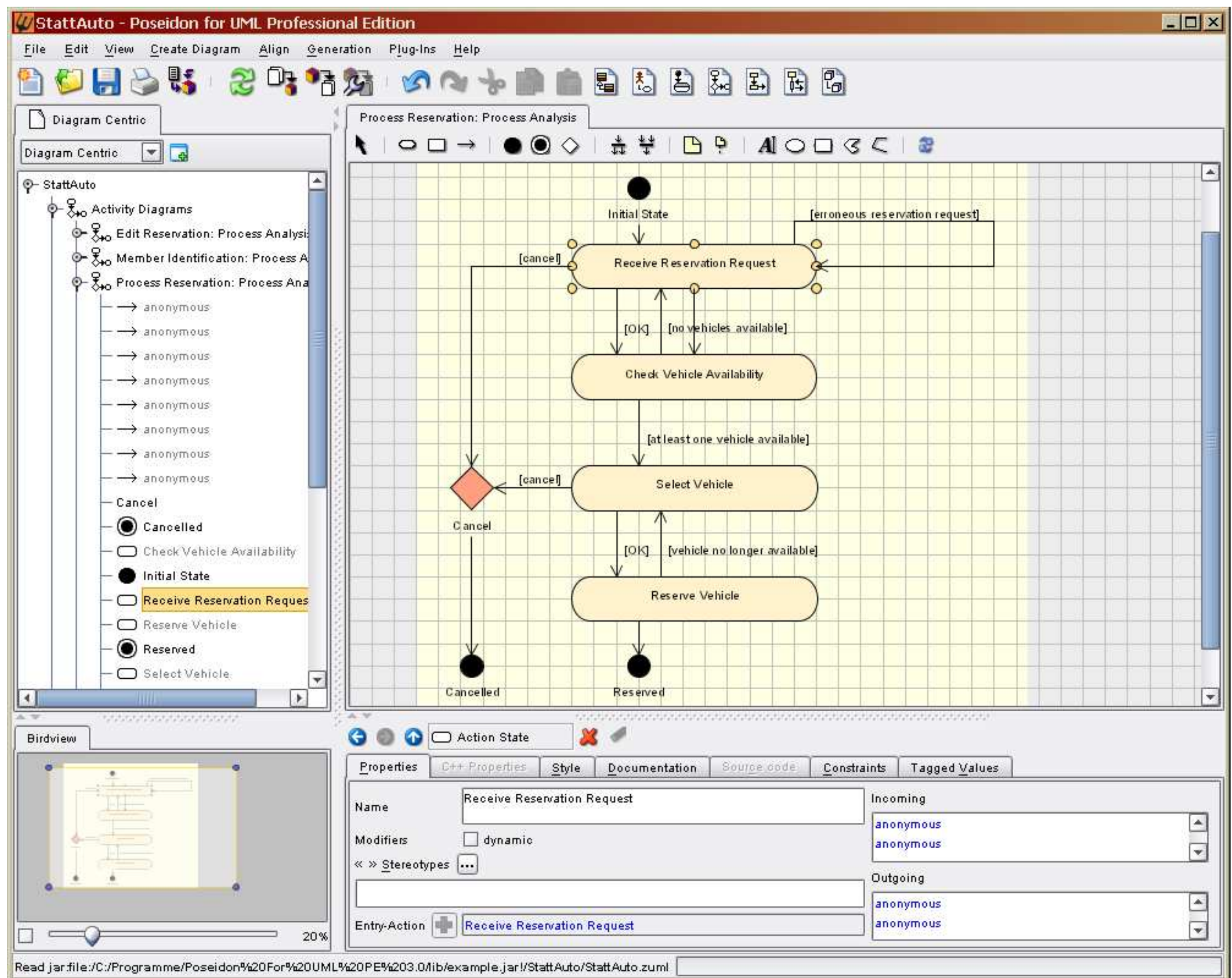


Figura A.2: (Cont.) Posseidon: Diagrama de Atividades.

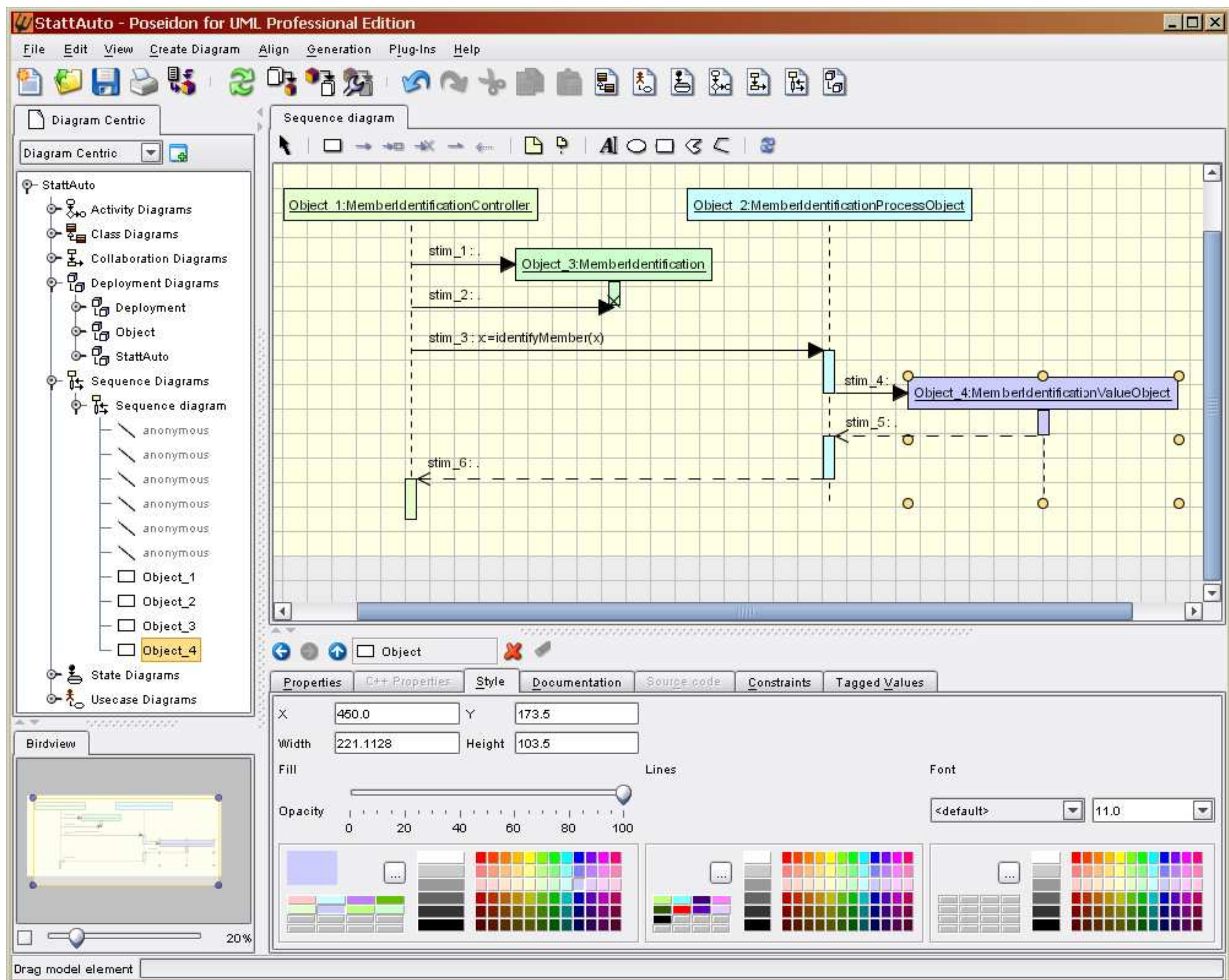


Figura A.2: (Cont.) Posseidon: Diagrama de Sequências.

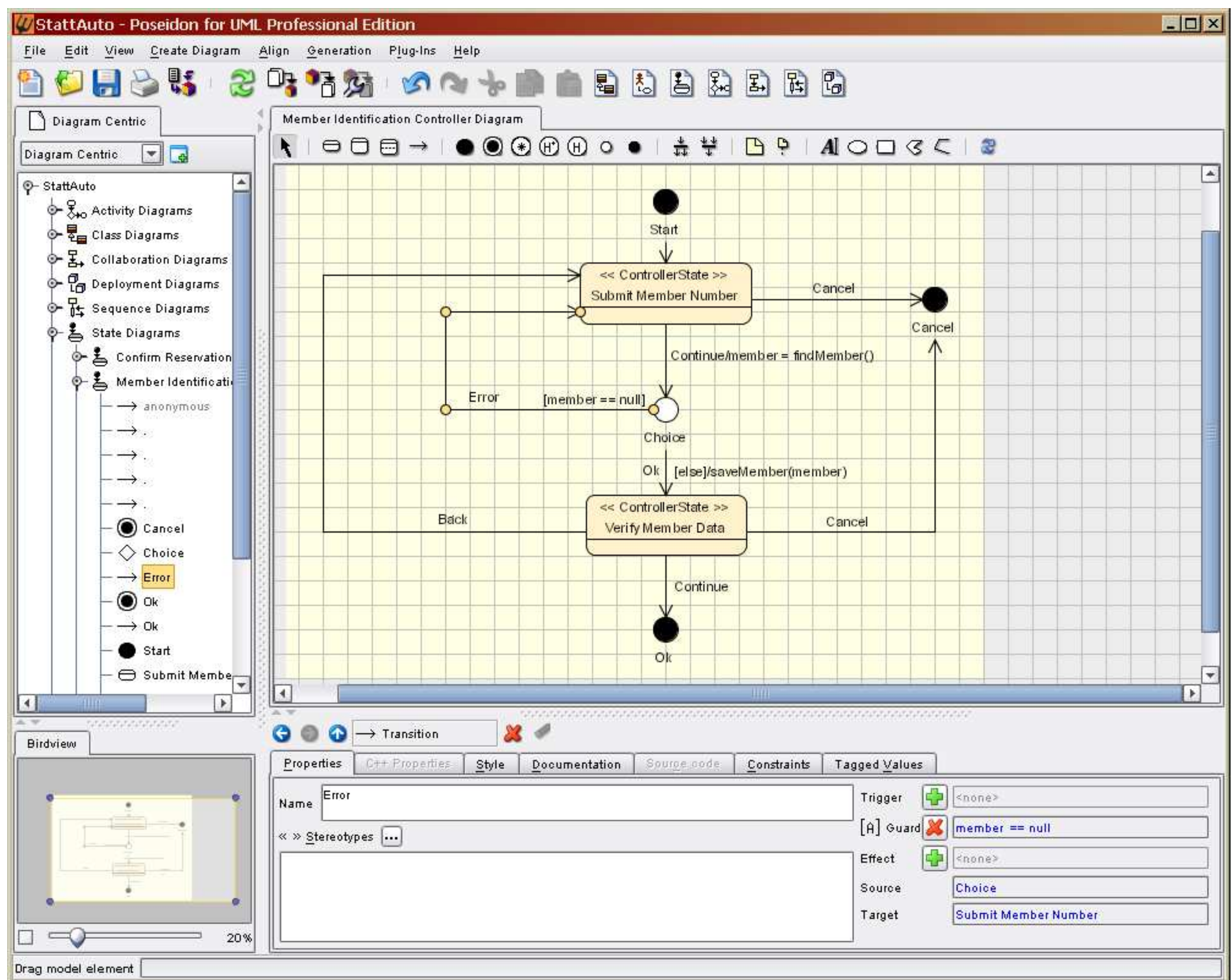


Figura A.2: (Cont.) Posseidon: Diagrama de Estados.

Vale lembrar que essas duas ferramentas são feitas em Java, logo requerem um pouco de memória e uma máquina virtual Java JDK 1.4 instalada. Mas podem ser utilizadas em qualquer plataforma (Unix, Linux, Mac OSX, Windows). Outro ponto a se destacar é que estas ferramentas ocupam pouco espaço em disco e podem ser utilizadas livremente. Talvez sejam uma boa alternativa a ferramentas absurdamente caras como o Rational Rose e/ou extremamente pesadas como a Thogether for Java.

Apêndice B

Erros Comuns de Compilação

A seguir, uma lista dos erros de compilação mais comuns e sua possível correção.



	Exemplo (com erro)	Correção
1.	Esquecendo o ";"	

	`#include <stdio.h>`
	`void main(){`
	`printf("Oi")`
	`}`
		`#include <stdio.h>`
		`void main(){`
		`printf("Oi");`
		`}`

	Error TESTE.CPP 4: **Statement missing ;** Error TESTE.CPP 4: **Compound statement missing }**	Erro: declaração esquecida: '`;`', ou seja, o programador simplesmente esqueceu do ponto-e-vírgula Significa que o compilador não encontrou o final do programa. Esperava o '`;`' antes do '`}`'. Este erro foi gerado como efeito "cascata" do erro anterior. Pode ser ignorado.
2.	Falta incluir bibliotecas: #include<???.h>	

	`#include <stdio.h>`
	
	`void main(){`
	`clrscr();`
	`printf("Oi");`
	`}`
		`#include <stdio.h>`
		`#include <conio.h>`
		`void main(){`
		`clrscr();`
		`printf("Oi");`
		`}`

	Error TESTE.CPP 3: **Function 'clrscr' should have a prototype**.	A mensagem de erro se refere à função '`clrscr()`' que não foi declarada. No caso, faltou no cabeçalho do programa, incluir a biblioteca onde esta função é definida: `#include <conio.h>`.
3.	Confundindo "," com "." – informando incorretament casas decimais no C	

	`#include <stdio.h>`
	`void main(){`
	`float r= 3,5;`
	`}`
		`#include <stdio.h>`
		`void main(){`
		`float r= 3.5;`
		`}`

	Error TESTE.CPP 3: **Declaration terminated incorrectly** **Warning** TESTE.CPP 4: 'r' is **assigned a value that is never used**	Erro: declaração terminada de forma incorreta, isto é, o usuário digitou '`3,5`' quando deveria ser '`3.5`'. Advertência: a variável 'r' foi declarada mas nunca usada. Esta é apenas uma advertência. Não significa erro de programação.

4. Erro de sintaxe na declaração – falta do operador **#**

```
_ include <stdio.h>
void main(){
  printf("Teste");
}
```

```
#include <stdio.h>
void main(){
  printf("Teste");
}
```

Erro	Explicação
Error TESTE.CPP 1: **Declaration syntax error**	Declaração com erro de sintaxe. O programador esqueceu o "#" na frente do `include`.

5. Fechando printf() de maneira incorreta, usar: **"**

```
#include <stdio.h>
void main(){
  printf("Teste’_ );
}
```

```
#include <stdio.h>
void main(){
   printf("Teste");
}
```

Erro	Explicação
Error TESTE.CPP 3: **Unterminated string or character constant**	String não terminada; faltou "`"`" ao invés de "`’`".
Error TESTE.CPP 4: Function call missing)	O compilador não reconheceu o "`)`" do final de função `printf()`. Efeito cascata do primeiro erro.
Error TESTE.CPP 4: Statement missing ;	O compilador não reconheceu o ";" – efeito cascata do primeiro erro.
Error TESTE.CPP 4: Compound statement missing }	O compilador não encontrou o final do programa. Não reconheceu o "`}`" no final do programa – efeito cascata do primeiro erro.

6. Uso incorreto de "**(**" e "**)**"

```
#include <stdio.h>
void main(){
  printf _"Teste"_;
}
```

```
#include <stdio.h>
#include <stdio.h>
  printf ("Teste");
}
```

Erro	Explicação
Error TESTE.CPP 3: **Statement missing ;**	O compilador esperava um "`;`" antes de encontrar o início da mensagem para impressão "`"`".

7. Separando incorretamente parâmetros de um função, usar "**,**")

```
#include <stdio.h>
void main(){
  int a=4;
  float b=1.5;
  printf ("a + b = %d + %f = %f"; a, b,
a+b);
}
```

```
#include <stdio.h>
void main(){
  int a=4;
  float b=1.5;
  printf ("a + b = %d + %f = %f", a, b,
a+b);
}
```

Erro	Explicação
Error TESTE.CPP 5: **Function call missing)**	Faltou o "`)`" na declaração da função. Isto é, o compilador imaginou que a função `printf()` terminasse desta forma: `printf ("a + b = %d + %f = %f");`.
Error TESTE.CPP 5: Statement missing ;	Idem ao anterior, mas neste caso o compilador esperava o "`;`" para indicar o fim da função `printf()`.

ERROS DURANTE A EXECUÇÃO DO PROGRAMA (*run-time errors*):

8. Falta do operador **&** com a função scanf()

```
#include <stdio.h>
void main(){
  float a;
  printf("a ?  ");
  scanf("%f", a);
  printf ("a = %5.2f\n",a);
}
```

Sem erros de compilação, mas...
Saída do programa:

a ? **scanf:** floating point **formats not linked**
Problema de associação de dados com scanf()

Abnormal program termination

```
#include <stdio.h>
void main(){
  float a;
  printf("a ?  ");
  scanf("%f", &a);
  printf ("a = %5.2f\n",a);
}
```

ERRO: foi informado na função `scanf()` que seria lida uma variável do tipo *float*, mas não foi especificado de forma correta a variável com a qual a entrada de dados estaria ligada. Faltou o "`&`" na linha com `scanf()`.
Término anormal do programa. Consequência do erro anterior.

9. Esquecendo-se de terminar a mensagem dentro de printf()

```
#include <stdio.h>
#include <conio.h>
void main(){
  int a=4;
  float b=1.5;
  clrscr();
  printf ("a + b = %d + %f = %f , a, b,
a+b");
}
```

Warning TESTE.CPP 8: 'b' is **assigned a value that is never used**
Warning TESTE.CPP 8: 'a' is assigned a value that is never used

Saída do programa:

printf: floating point **formats not linked**

Abnormal program termination

```
#include <stdio.h>
#include <conio.h>
void main(){
  int a=4;
  float b=1.5;
  clrscr();
  printf ("a + b = %d + %f = %f", a, b,
a+b");
}
```

Advertência: variável 'b' recebeu valor que nunca foi utilizado.
Advertência: variável 'a' recebeu valor que nunca foi utilizado.

A função `printf()` incluia a impressão de variáveis do tipo ponto flutuante ("`%f`") que no entando nunca foram indicadas ao final de `printf()`.
O erro anterior provocou o término anormal do programa – na realidade, falha na execução do `printf()` anterior.

Índice Remissivo